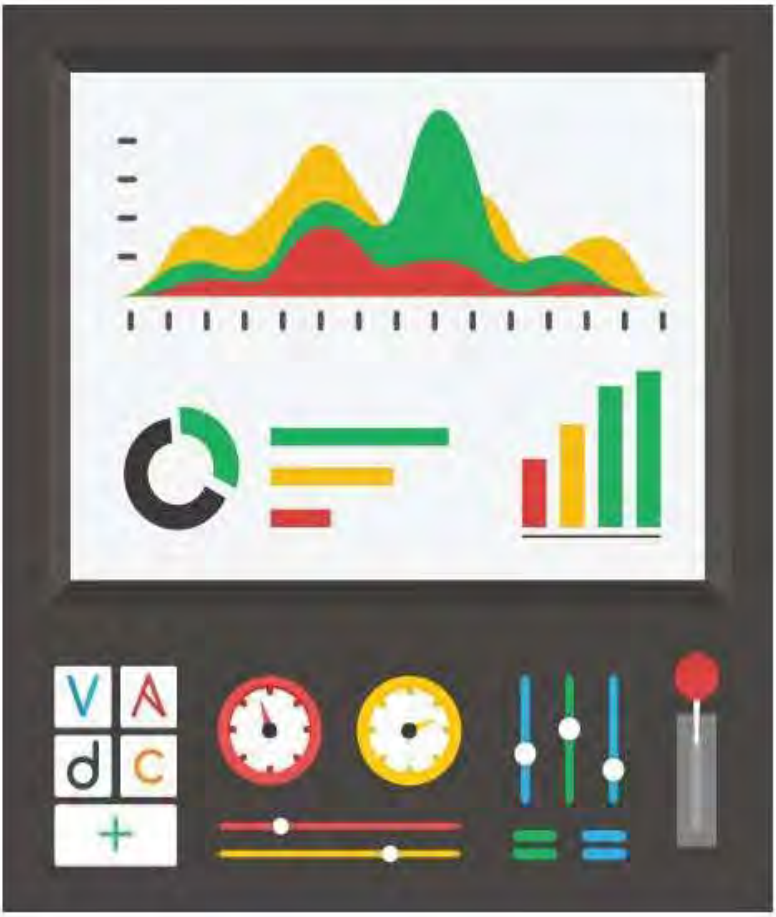
Caixa de Ferramentas DevOps

Um guia para construção, administração e arquitetura de sistemas modernos



Casa do Código

GLEICON MORAES

SOBRE O AUTOR

Passei minha infância e adolescência mexendo em eletrônica, guitarra e computadores 8 bits. Hoje sou formado em Ciência da Computação e pós-graduado em Sistemas de Informação.

Gosto muito do que faço: construo e conserto sistemas distribuídos e de larga escala há 20 anos. Além de programar em Python, Erlang e Go, faço café, e-mail, cloud, big data e automação de infra.

Há algum tempo comecei a trabalhar gerenciando pessoas e me apaixonei pela possibilidade de criar equipes e conquistar grandes projetos. Quando descobri minha carreira gerencial tive certeza de que deveria me esforçar mais para continuar relevante tecnicamente e falar a língua das pessoas que trabalham comigo. Tive a sorte de participar de empresas e equipes que desenvolvem e operam alguns dos maiores sistemas da internet do Brasil.

Escrevi um livro em 2005 chamado *Programação Avançada em Linux*, o primeiro livro brasileiro a falar sobre kernel, módulos, drivers, dispositivos eletrônicos e temas avançados do sistema operacional.

Já palestrei em várias edições da RubyConf, QCon e OSCon, no Brasil e nos Estados Unidos. Tenho um repositório de código (https://github.com/gleicon) e publico o que acho interessante no Twitter (https://twitter.com/gleicon).

Meu perfil profissional no Linkedin fica em https://linkedin.com/in/gleicon. Tenho um site com links para projetos em http://7co.cc/. O material das palestras que já dei pode ser encontrado em http://www.slideshare.net/gleicon e também em https://speakerdeck.com/gleicon/.

Participe do nosso grupo de discussão do livro, em https://groups.google.com/forum/#!forum/caixa-de-ferramentas-devops.

INTRODUÇÃO

Este livro é uma introdução com opiniões sobre ferramentas para desenvolvimento e administração de sistemas. As ferramentas demonstradas são flexíveis e extensíveis, o que permite que a mesma tarefa seja executada de formas distintas.

Meu objetivo é mostrar uma maneira de utilizá-las para ganhar produtividade rapidamente. Ao longo do texto, coloquei referências para que o leitor possa se aprofundar ou buscar um conceito teórico que está fora do escopo proposto. Não vou me deter em discussões holísticas de implantação de conceitos Ágeis ou DevOps.

Originalmente, este livro era um conjunto de notas que fui colecionando durante o dia a dia e conversas com colegas. Revisando estas notas quando precisei começar um novo projeto me dei conta de que elas contavam uma história interessante para quem teve pouco ou nenhum contato com ferramentas de automação e virtualização. Mais ainda, elas me ajudaram a treinar outras pessoas com pouco tempo e com objetivos maiores do que se especializarem em Ansible, Vagrant ou Virtualbox.

As ideias descritas podem ser implementadas e utilizadas com qualquer substituto destas ferramentas — provavelmente você tem um `deploy.sh` em algum diretório ou repositório que faz mais que elas em conjunto. Isso é bom, pois mostra que a necessidade existe e que já foi investido um tempo em atendê-la. Minha proposta neste caso é explorar a combinação das ferramentas apresentadas para entender a maneira modular como o mesmo problema é resolvido por elas.

Automatização é um amplificador da energia que você investe em suas tarefas. O mesmo argumento que era utilizado para controle de versão pode ser utilizado para automação: você

compromete um tempo aprendendo, investe um pouco mais nos primeiros passos e depois ganha em escala e qualidade de trabalho.

Uma das ideias que vou explorar no texto é de que todo repositório de código tenha uma estrutura mínima que consiga criar um ambiente para desenvolvimento ou teste localmente. Ao montar uma estrutura de automação para sua aplicação que funciona localmente, você ganha a mesma estrutura para seu ambiente de produção.

Desenvolver localmente com a habilidade de criar ambientes com arquitetura semelhantes às encontradas em ambiente de produção é um principio poderoso. Ele habilita o desenvolvimento incremental e testes funcionais, além da familiaridade com a arquitetura do sistema.

A velocidade de desenvolvimento e avaliação de bibliotecas e projetos de código aberto também aumenta com a habilidade de criar um ambiente isolado com todas as dependências e descartá-lo após o uso. Finalmente, é um treinamento para o processo de *deploy*.

Se encararmos o fato de que sistemas e suas arquiteturas não são estáticos, é importante desenvolver um conjunto de práticas para acompanhar o desenvolvimento e mudança desta arquitetura. Temos que ter respostas para recriar o ambiente em caso de desastres, fazer crescer seus componentes quando confrontados com uma carga inesperada e desenvolver em uma réplica em escala do ambiente final.

Ferramentas como Ansible e Vagrant são associadas ao movimento DevOps, que explora mudanças culturais e organizacionais, além de novas abordagens para problemas conhecidos como gerenciamento de configuração, monitoramento e coletas de métricas, aplicação de técnicas de engenharia de software

na criação de infraestrutura e a interação rápida entre equipes.

Acompanhe podcasts (por exemplo o FoodFight, listas como DevOps Weekly, Reddits como reddit.com/r/devops e reddit.com/r/sysadmin e conferencias como a Velocity para novas ideias e projetos.

Com as ferramentas que vamos ver, é fácil testar de forma controlada novas ideias apresentadas nestes canais. Não é mandatório adotar ou se associar a qualquer grupo ou metodologia para ter benefícios. São boas práticas colhidas e compartilhadas por pessoas com experiência de sistemas em produção. O custo é baixo e o maior investimento é o tempo para consumir o material disponível.

Vou assumir que seu sistema operacional é baseado em Linux ou MacOS X. Usuários de Windows podem seguir as mesmas instruções, mas provavelmente terão que trocar a direção da barra (de / para \ ), encontrar um editor de textos legal e pensar um pouco onde declarar variáveis de ambientes e bibliotecas de Python. O código demonstrado neste livro estará disponível em um repositório em minha conta no GitHub (https://github.com/gleicon/caixa-de-ferramentas-devops). Este código é livre exceto aonde a licença das bibliotecas utilizadas informem.

Para as maquinas virtuais e receitas de Ansible, vou utilizar a última versão LTS de Ubuntu Server. Vou marcar os pontos específicos e explicar um pouco de como fazer o mesmo para distribuições baseadas em RPM — é realmente fácil e mostra como vale a pena investir tempo em um framework de automação como o Ansible. Também vou assumir que você consegue usar o terminal (Terminal, iTerm, RXVT, XTerm etc.) disponível. Novamente, para Windows vou ficar devendo algo além do `command prompt` e da versão do PuTTY que funciona localmente. Se souberem de uma alternativa legal que não seja instalar cygwin + XWindow me

contem.

Por último, tentei cobrir exemplos de aplicações simples e repeti-los sob condições diferentes para criar uma história de evolução do uso das ferramentas.

Vamos instalar um Wordpress em várias configurações e provedores de serviço e posteriormente um Banco de dados NoSQL chamado Cassandra. São aplicações que cobrem alguns padrões de uso e configuração que se repetem. Com isso, quero estimular a discussão e a reflexão da arquitetura de suas aplicações. Pense em como elas são compostas e qual a melhor maneira de organizá-las.

Enjoy!

Sumário

1 Linux, SSH e Git

1.1 Linux 1

1.2 SSH 4

1.3 Git 7

1.4 Conclusão 14

2 Vagrant

2.1 Introdução 15

2.2 Caixa de Ferramentas — Vagrant e VirtualBox 16

2.3 Instalação do VirtualBox 17

2.4 Instalação do Vagrant 20

2.5 Conclusão 27

3 Ansible

3.1 Instalação 29

3.2 A configuração do Ansible e o formato YAML 29

3.3 Ansible e Vagrant 31

3.4 Refatoração 34

3.5 Conclusão 36

4 Instalando WordPress em uma máquina

4.1 Expandindo nosso playbook — Um blog com WordPress 38

5 Proxy Reverso e WordPress em duas máquinas

5.1 nginx e proxy reverso 52

5.2 WordPress em duas máquinas 57

5.3 Instalando seu WordPress em máquinas virtuais. 60

5.4 DigitalOcean API v2 68

5.5 Conclusão 70

6 Cassandra e EC2

6.1 Vagrant e Cassandra 73

6.2 Provisionamento na AWS 77

6.3 Cassandra e AWS 83

6.4 Conclusão 87

7 Métricas e monitoração

7.1 Monitoração 89

7.2 CollectD e Logstash 94

7.3 Profiling 98

7.4 Conclusão 100

8 Análise de performance em cloud com New Relic

8.1 WordPress na AWS 102

8.2 Instalando o New Relic 103

8.3 Gerando carga — profiling da aplicação 105

8.4 Simulando limitações de rede 112

8.5 Conclusão 118

9 Docker

9.1 Virtualização 120

9.2 Containers e CGroups 122

9.3 Docker 125

9.4 Boot2Docker 134

9.5 Criando nosso container de PHP e NGINX e usando o Docker Hub 135

9.6 Docker Compose 141

9.7 Conclusão 151

10 Em produção

10.1 Deploy 153

10.2 Building Blocks e Lock-In 154

10.3 Blue/Green Deploy e pacotes 158

10.4 Testes 159

10.5 Conclusão 164

Capítulo 1

LINUX, SSH E GIT

Neste capítulo vamos ver um resumo de três assuntos que permeiam o livro: Git, SSH e Linux. Se você é fluente nestas tecnologias pode pular esta introdução tranquilamente.

Estes assuntos em si criaram livros bem completos e interessantes. Vou demonstrar e explicar o necessário para sobreviver aos próximos capítulos. Você deve procurar outros livros e material na internet para se aprofundar caso tenha interesse.

1.1 LINUX

O sistema operacional utilizado em todos os exemplos deste livro é o GNU/Linux, distribuição Ubuntu na sua versão 14.04 LTS de 64 bits. Com poucas adaptações, outras distribuições podem ser utilizadas. Escolhi reduzir as opções para nos concentrarmos no objeto do livro, principalmente quando falarmos sobre Ansible.

Assumi também que sua máquina tem alguma variante de Linux ou Mac OS X. Falei um pouco sobre isso na introdução. Não vamos utilizar comandos complexos, mas vou utilizar o *Terminal* sempre que possível. No Mac OS X eu uso o *ITerm 2* mas nada impede de utilizar o *Terminal.app*. No Linux fique à vontade para usar o terminal com o que está acostumado.

O interpretador de shell que uso é o *bash* que estiver instalado, e o editor de textos é o *Vim*, também na versão que estiver disponível.

Onde for preciso, fornecerei listagens de códigos para serem digitadas e cópias em um repositório do GitHub.

Esta versão de Linux é encontrada nos provedores de serviço que vamos utilizar, o que facilita a transposição dos exemplos para fora da máquina local. Explore as diferenças entre um Linux executando localmente e remotamente. Crie máquinas e observe o comando `top` nelas.

Aconselho que digite os exemplos no ambiente Linux para se familiarizar com as ferramentas e comandos. Nos próximos capítulos vou sugerir que o que fizermos localmente seja feito em uma máquina em um provedor de *cloud* externo. A progressão dos capítulos nos levará a criar as máquinas automaticamente, mas enquanto não chegamos lá, você precisará de máquinas criadas com autenticação por chave de SSH.

Adiante, veremos como criar máquinas na AWS (http://aws.amazon.com) e na DigitalOcean (http://digitalocean.com). Escolhi estes dois provedores pois eles têm cupons e opções de criar e utilizar máquinas sem custo por um período limitado, o que é ideal para o aprendizado.

Qualquer produto usado em provedores de cloud tem um custo. *Load balancer* são cobrados por tráfego, e volumes de storage são cobrados por tráfego e armazenamento. Máquinas virtuais geralmente são cobradas enquanto estão ligadas ou enquanto existirem. Leia bem a documentação oferecida e as regras de cobrança.

Na DigitalOcean, o processo é simples: procure no Twitter deles (@DigitalOcean) um código ou link e crie uma conta. Às vezes, na primeira vez que você cria uma conta já ganha um crédito de 5 dolares, o suficiente para um mês contínuo da menor máquina virtual que eles oferecem. Se utilizarmos mais máquinas por um

periodo menor de tempo, criando e destruindo as instâncias (outro nome para máquina virtual) este valor dá e sobra.

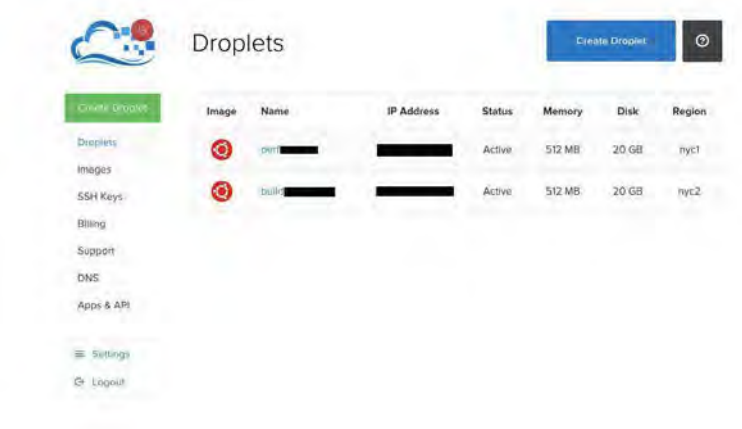


Figura 1.1: Painel de criação de máquinas da DigitalOcean

Na AWS, se você já não tem uma conta, precisa criar, configurar um VPC (*Virtual Private Cluster*), um *security group* e *subnet*. Você consegue executar uma máquina sem isso apenas entrando no console, selecionando a opção `EC2` e no painel que aparecer clicar no botão a seguir:

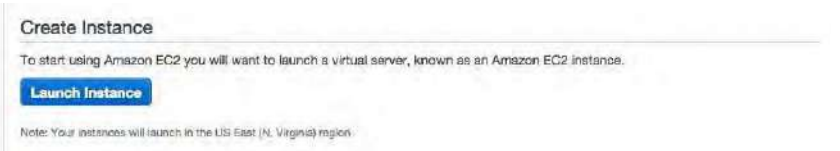


Figura 1.2: Botão de criação de máquinas na AWS

O processo de criação e utilização da AWS é complexo pois oferece toda a infraestrutura de um datacenter de forma virtual e utilizável programaticamente via API. Leia a documentação em (https://aws.amazon.com) para entender melhor e use o *free tier* para seguir os exemplos oferecidos.

Nas opções de criação de instâncias serão feitas várias perguntas. É seguro inicialmente usar as opções padrão. *Lembre-se sempre de desligar as máquinas no console para evitar cobrança extra*.

1.2 SSH

O SSH é o protocolo utilizado para se conectar de forma segura em servidores. Por meio de criptografia, ele cria um canal seguro entre duas máquinas. O SSH pode ser utilizado também para executar comandos remotamente sem entar no *shell* da máquina.

A configuração do SSH é rica, mas para os propósitos deste livro é importante aprender uma técnica de autenticação apenas. Normalmente, quando nos conectamos a uma máquina aparece um pedido de senha.

Para automatizar o gerenciamento de configuração precisamos conectar aos servidores sem este pedido de senha de forma segura. Provedores de cloud como a Amazon oferecem esta opção como a principal e recomendada para o acesso a servidores. Outros sistemas de gerenciamento de configuração preveem a instalação de um agente nas máquinas. O Ansible só precisa deste acesso por SSH.

Na maioria dos provedores, você encontrará uma opção para adicionar uma chave de SSH pública (*Public SSH Key*). Na AWS, o caminho das chaves fica dentro do item `EC2` no subitem `Key Pairs`.

Você pode criar suas chaves no console e um arquivo com a extensão `.pem` será salvo em seu computador. Na DigitalOcean, você deve adicionar uma chave existente e o gerenciamento é diferente da AWS mas o resultado é o mesmo: as máquinas criadas terão esta chave configurada automaticamente.

O painel de gerenciamento e criação de chaves da AWS:

Figura 1.3: Gerenciamento de chaves na AWS

O painel de gerenciamento de chaves da DigitalOcean:

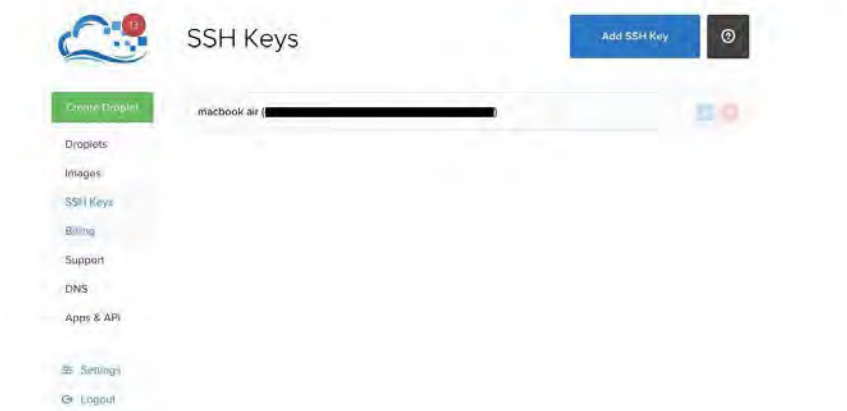


Figura 1.4: Gerenciamento de chaves na DigitalOcean

Vamos gerar um par de chaves localmente. No terminal digite:

```
$ ssh-keygen -t rsa
```

No momento em que o comando perguntar sobre uma *passphrase*, digite `[ENTER]` duas vezes sem entrar o passphrase. O resultado esperado é o da próxima figura:

Figura 1.5: ssh-keygen

Nossa chave foi criada com o nome `id_rsa_devops` e dois arquivos foram criados dentro do diretório `ssh` no home do seu usuário: `id_rsa_devops` e `id_rsa_devops.pub`.

Com o par de chaves pública e privada podemos configurar o acesso remoto a outras máquinas. O arquivo com extensão `.pub` é o único que deverá ser copiado para outras máquinas e para painéis de configuração. Para configurar a chave na DigitalOcean, siga este tutorial (https://www.digitalocean.com/community/tutorials/how-to-use-ssh-keys-with-digitalocean-droplets) a partir do três. É uma sequência de telas indicando onde você coloca a chave pública criada dentro do painel de controle.

O teste deste tutorial é criar uma máquina e executar o comando:

```
$ ssh root@ip-da-maquina
```

A chave será colocada no usuário `root`. Se você já tem uma máquina criada ou gostaria de fazer login com outro usuário que não seja o `root`, faça uma copia da sua chave úublica, entre na

máquina pelo método normal de conexão com senha e crie uma entrada no arquivo `authorized_keys` que fica dentro do diretório `.ssh/` com o conteúdo do arquivo que tem sua chave pública — no nosso caso, `id_rsa_devops.pub`.

1.3 GIT

O Git é um sistema de controle de versão gratuita. Seu site fica em (http://git-scm.com/). Em termos práticos, sua função é controlar mudanças em repositórios de dados. Eu digo dados porque ele pode ser usado para outros tipos de dados além de codigo-fonte de programas.

Este livro, por exemplo, foi escrito utilizando um sistema baseado no (https://www.gitbook.com), que utiliza o Git para controlar versões e o progresso do código-fonte de um livro.

É uma ferramenta simples mas bem pensada, portanto alguns conceitos e ideias podem demorar para fazer sentido para quem não usou um sistema de controle de versão. Mas é o tipo de ferramenta que você aprende muito rápido quando a inclui em seu dia a dia.

Uma das formas de utilizar o Git em seus projetos é com o GitHub em (https://github.com). Vamos instalar o Git em seu sistema operacional. No site do Git e do GitHub você encontrará instruções mais detalhadas. Minha recomendação é utilizar o gerenciador de pacotes do seu sistema operacional:

```
# Para linux baseado em debian:
$ sudo apt-get install git-core

# Para linux CentOS/RH
$ sudo yum install git

# Para MacOS X com homebrew
$ brew install git
```

Para Windows, procure uma versão compatível no site (http://git-scm.com/downloads). Existem interfaces gráficas para utilizar o Git mas vou me manter no terminal. Você pode utilizá-las se está acostumado.

Após instalar o Git, execute os comandos a seguir:

```
git config --global user.name "Seu nome"
git config --global user.email "seu@email.com.br"
```

O Github é um site que fornece repositórios remotos de Git em sua conta sem custo, caso o projeto seja aberto, e com custo caso você queira repositórios privados. A diferença é que repositórios privados só serão vistos por pessoas que você autorizar. Para utilizá-lo, vá até o site e crie uma conta. O processo é simples e você será redirecionado ao seu painel de controle, que, se não mudou até a publicação deste livro, é assim:

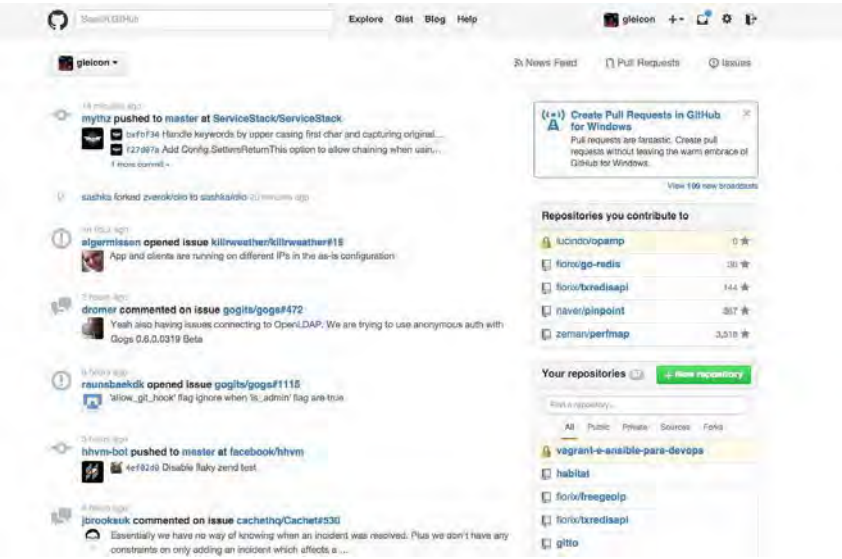


Figura 1.6: Painel do Github

Neste ponto, precisamos de um par de chaves para o SSH. O GitHub tem uma boa explicação para este procedimento em

(https://help.github.com/articles/generating-ssh-keys/) sob a perspectiva de sua configuração. Com o que vimos anteriormente não deve ser um problema. Você pode copiar a chave criada (`~/.ssh/id_rsa.pub`) diretamente no passo 4 se já tem sua chave criada. Se não criou anteriormente, siga este tutorial.

Com a chave configurada, vamos criar um novo repositório. Note um botão chamado `+ New Repository` do lado direito, ao lado do texto *Your repositories*. Clique nele.

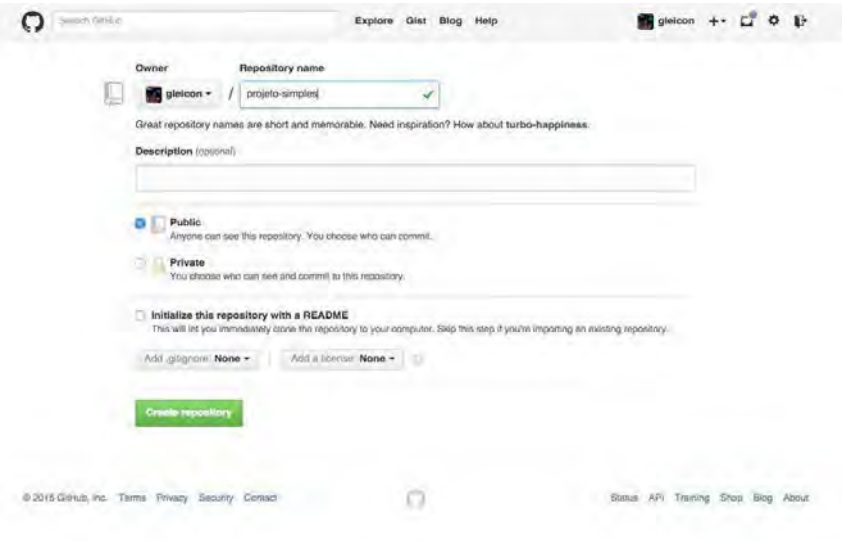


Figura 1.7: Criação de repositório

Eu preenchi a caixa embaixo do texto *Repository Name* com o nome `projeto-simples` . Deixei o resto das opções como estavam e cliquei em `Create Repository` .

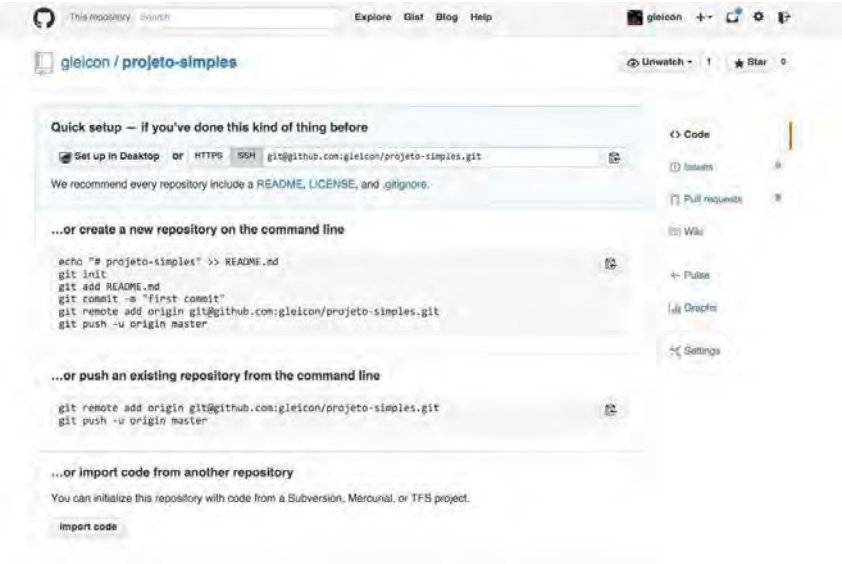


Figura 1.8: Repositório criado

Na barra de endereços do seu browser está o endereço público do seu repositório. No lugar do usuário *gleicon* aparecerá seu usuário. Como nenhum código foi enviado ainda, a tela que aparece é uma ajuda para a configuração de um novo repositório ou para a ativação de um novo *remote* em um repositório existente.

Remote, na terminologia do Git, é um repositório remoto. Vamos criar um repositório local que conhece este repositório remoto e mediante comandos sincroniza ou envia as mudanças para lá.

Esta troca de dados é bilateral. Podemos recuperar mudanças em um repositório remoto para o repositório local, por exemplo quando um colega tem autorização ao repositório e modifica o código. O Git chama estas transferências de *deltas*.

Abra o terminal do seu sistema operacional e crie um diretório chamado `projeto-simples` :

```
$ mkdir projeto-simples
```

Entre neste diretório com `cd projeto-simples` e use seu editor de textos para criar um arquivo chamado `README.md`. Neste arquivo digite o texto a seguir:

```
# README do meu projeto-simples

Este projeto é apenas um shell script que conta itens unicos no
diretório /etc
```

Crie um arquivo chamado `itens_unicos.sh`:

```
#!/bin/sh

echo "Itens unicos"
ls /etc | cut -d' ' -f 1 | sort | uniq | wc -l
```

Vamos seguir as instruções do GitHub, com uma pequena mudança em `git add`:

```
git init
git add .
git commit -m "first commit"
git remote add origin git@github.com:gleicon/projeto-simples.git
git push -u origin master
```

No browser, faça reload da página do seu repositório. Ela deve se parecer com a próxima imagem:

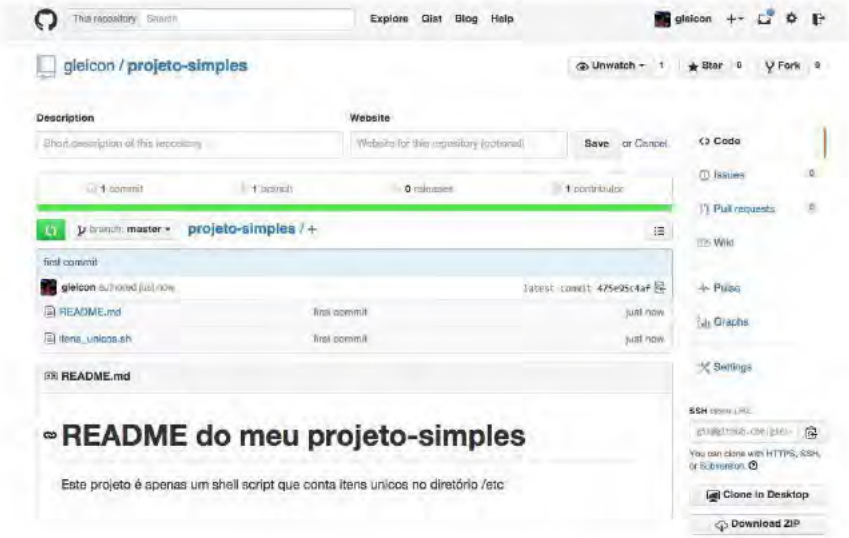


Figura 1.9: Repositório com código

Explore a interface do GitHub e sua documentação. Faça um teste em outro diretório ou usuário da mesma máquina (ou se puder em outra máquina que tenha o cliente Git instalado). Note no lado direito inferior da imagem uma caixa de texto acima de *Clone in Desktop* chamada *SSH clone URL*. Copie o valor desta caixa e execute:

```
$ git clone git@github.com:gleicon/projeto-simples.git
```

Após este comando, você deve ter uma réplica do repositório. Se houver um erro, pode ser que você esteja em uma máquina em que sua chave SSH não funcione. Execute o comando de uma forma mais simples:

```
git clone https://github.com/gleicon/projeto-simples.git
```

Para a maioria das tarefas, você deve utilizar esses comandos. Segue uma explicação breve deles:

- git init — Inicializa um repositório Git no diretório atual.

- git add . — Adiciona um ou mais arquivos (com . adiciona todos) para serem enviados (commit) ao repositório. Pode ser adicionados arquivos novos ao repositório e também aqueles que já estão adicionados mas foram modificados. O conjunto de adições, remoções e movimentações compõe um *commit*.
- git commit -m "mensagem" — Confirma as mudanças e cria um *commit* que é uma das unidades de trabalho do Git
- git remote ..... — Adiciona um remote no repositório atual, chamado `origin`. Você poderia trabalhar sem ter um remote, não é mandatório
- git push -u origin master — Envia (*push*) as modificações para o repositório remoto. O parametro `-u` só é necessário na primeira execução
- git clone — Clona o repositório dado pela URL localmente

Vamos modificar nosso arquivo, fazer um commit e um push para o repositório remoto. Crie o arquivo `portas.sh` e digite:

```
#!/bin/sh

echo "Lista de porta 80 no netstat"
netstat -an | grep 80
```

O passo a passo para adicionar este arquivo ao repositório é simples:

- git add portas.sh
- git commit -m "add portas.sh"
- git push origin master

Para fixar este processo, vamos aprender um novo comando para seu repositório :

```
$ git status
```

Este comando mostra o estado atual do repositório e das mudanças. A saída deste comando tem uma sessão chamada *Untracked files* e o arquivo `portas.sh` estará lá com uma indicação do comando `git add <file>` para adicioná-lo.

```
$ git add portas.sh
```

Agora o *git status* vai mostrar uma sessão chamada *Changes to be committed* com o arquivo `portas.sh` e uma indicação de uso do *git reset* para reverter a última mudança em relação ao estado do repositório. Vamos em frente com o comando `git commit` .

```
$ git commit -m "novo arquivo"
```

O parâmetro `-m` do comando `git commit` é apenas uma mensagem que indica o motivo da mudança. Após o commit, *git status* mostra uma mensagem indicando que seu *branch* local está um commit à frente do repositório `origin/master` e sugere o uso do `git push` .

```
$ git push origin master
```

Após o `git push` , a mensagem do git status é *Your branch is up-to-date with 'origin/master'. nothing to commit, working directory is clean*. Volte à pagina do seu repositório no GitHub e confira a mudança.

1.4 CONCLUSÃO

Com este conhecimento, você já está apto a usar o Git para seus projetos e a criar máquinas que podem ser acessadas sem senha, com o uso de chave pública de SSH. Se você já executa estas tarefas de outra maneira, não tem problema, os capítulos a seguir foram escritos de forma a funcionar com uma variação grande dos processos descritos.

Capítulo 2

VAGRANT

2.1 INTRODUÇÃO

O primeiro passo para criar um ambiente de produção ou desenvolvimento é ter uma máquina dedicada ou um espaço em uma máquina. Com a popularização de técnicas de virtualização, há algum tempo você pode ter máquinas virtuais executadas em sua máquina local.

Uma máquina física consegue executar mais de uma máquina virtual em paralelo. Aplicativos como VirtualBox, Parallels e VMWare são utilizados para gerenciar e executá-las.

Uma máquina virtual parada nada mais é que uma imagem de um disco e metadados que descrevem sua configuração: processador, memória, discos e conexões externas. A mesma máquina em execução é um processo que depende de um *scheduler* (agendador de tarefas e processos) para coordenar o uso dos recursos locais.

Você pode gerenciar suas máquinas virtuais diretamente das interfaces das aplicações citadas anteriormente ou pode utilizar bibliotecas e sistemas que abstraem as diferenças entre essas plataformas, com interface consistente para criar, executar, parar e modificar uma máquina virtual.

É importante na escolha deste sistema levar em conta que existem diferentes formatos de máquinas virtuais e diferentes

sistemas operacionais, portanto, um formato de template e alguns conceitos adicionais que auxiliem na criação destas máquinas são desejáveis.

Precisamos também de fontes de máquinas virtuais, inicialmente com o sistema operacional original, mas também com a possibilidade de ter máquinas customizadas. Após a criação da máquina, queremos executar um passo posterior, que chamaremos de **provisionamento**, para modificar o sistema operacional entregue, instalando pacotes e configurações, além de instalar sua aplicação e possivelmente salvar aquela versão da máquina virtual para uso posterior.

2.2 CAIXA DE FERRAMENTAS — VAGRANT E VIRTUALBOX

Para criar e gerenciar os ambientes de máquinas virtuais locais, escolhi o Vagrant (https //www vagrantup com/) Para executar as máquinas virtuais, escolhi o VirtualBox (https://www.virtualbox.org/).

O Vagrant gerencia e abstrai provedores de máquinas virtuais locais e públicos (VMWare, VirtualBox, Amazon AWS, DigitalOcean, entre outros). Ele tem uma linguagem específica (DSL) que descreve o ambiente e suas máquinas. Além disso, ele fornece interface para os sistemas de gerenciamento de configuração mais comuns como Chef, Puppet, CFEngine e Ansible.

Com o VirtualBox, temos uma alternativa de código aberto e grátis para executar as máquinas virtuais localmente. Ele é bem integrado com o Vagrant e tem todas as opções de outras plataformas. Por um preço módico você pode trocar por outro provedor e ainda assim utilizar a mesma linguagem e interface para gerenciar suas máquinas e ambientes da mesma forma.

2.3 INSTALAÇÃO DO VIRTUALBOX

Vá até o site do VirtualBox em (https://www.virtualbox.org), faça o download da última versão estável para seu sistema operacional. Ele deve pedir para instalar algumas extensões que permitem a execução de máquinas virtuais e packs de expansão para outros dispositivos como USB e volumes de disco. Você verá que quando executa o VirtualBox ele mostra um menu que permite criar máquinas a partir de imagens ISO ou importar máquinas existentes.

Se ocorrer algum problema nesta fase, verifique se o seu sistema operacional está atualizado, se a versão do VirtualBox é a correta, se tem espaço em disco e se já não tem uma versão instalada que possa estar em conflito. Leia a documentação de *troubleshooting* antes de postar um bug e, se encontrar algo novo, avise os desenvolvedores.

Neste ponto, você pode fazer o download de uma ISO de Ubuntu neste endereço (http://www.ubuntu.com/download/server) e testar a criação de uma máquina virtual.

Execute o VirtualBox e clique em `New` no canto superior esquerdo. Um diálogo que pede o nome e o tipo de sistema operacional aparecerá. Preencha com o nome da sua máquina virtual, sistema operacional e versão do sistema.

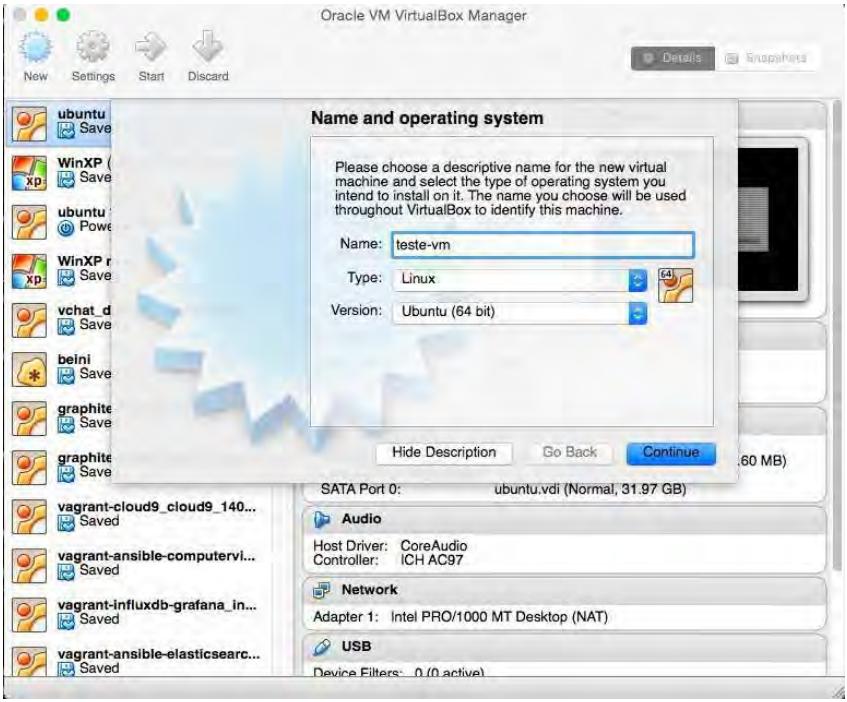


Figura 2.1: VirtualBox - criação da máquina virtual

Os próximos diálogos serão sobre tamanho da memória, tamanho e tipo de disco. Selecione a quantidade de memória de 1GB, indique que o disco será novo, dê um nome a ele, diga qual o espaço alocado e se cresce dinamicamente. Mantenha as opções *default* em caso de dúvida.

Sua máquina foi criada mas ainda está vazia. De um duplo clique em cima dela e execute (ou *Run* se estiver em inglês).

Figura 2.2: VirtualBox - primeira execução da máquina virtual

O diálogo a seguir aparecerá, com a tela preta de fundo. Neste ponto, selecione a imagem ISO de que fez download anteriormente e aguarde a instalação do sistema operacional. Finalize a instalação do Ubuntu (se tiver dúvidas, leia a documentação). Marque o tempo final e quantos minutos levou para instalar esta máquina. Esta informação será importante para a próxima sessão.

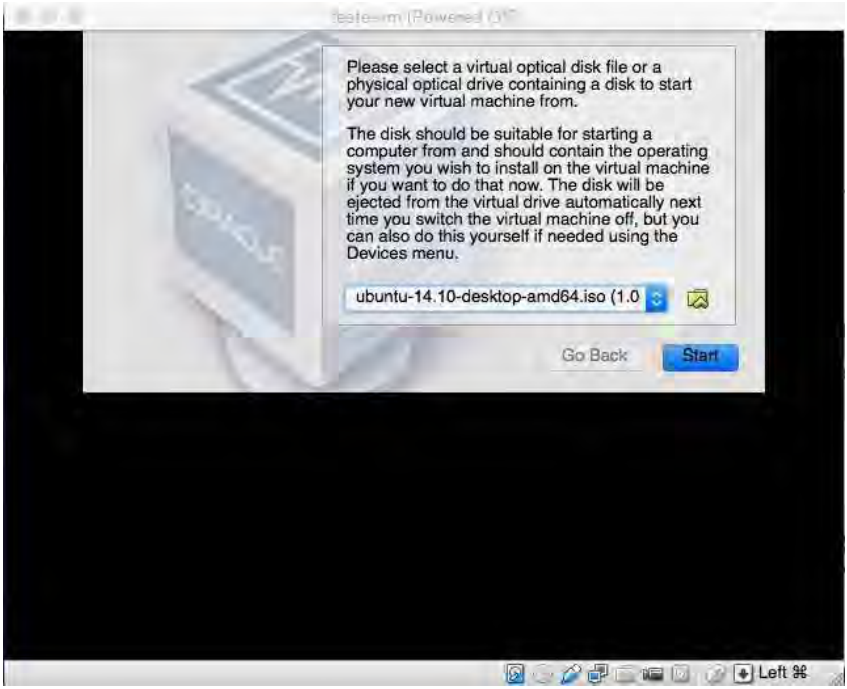


Figura 2.3: VirtualBox - instalação do sistema operacional

A instalação de máquinas virtuais em provedores de infraestrutura não é diferente desta sequência. Em alguns casos, algumas opções são predefinidas para poupar o usuário, por exemplo, imagens disponíveis.

2.4 INSTALAÇÃO DO VAGRANT

Selecione e faça o download da versão de Vagrant para seu sistema operacional em https://vagrantup.com. Leia a documentação pois há informações interessantes e sempre atualizadas das plataformas suportadas. Um instalador é fornecido e você tem tudo de que precisa em poucos cliques.

Abra seu terminal e digite `vagrant help` .

As opções apresentadas podem variar de acordo com a versão atual do Vagrant. Vamos começar com algumas opções comuns para gerenciar o ciclo de vida da máquina virtual.

Criaremos uma máquina usando o Vagrant. No terminal, crie um diretório chamado `testevm`. Entre nele e com seu editor de textos predileto (por exemplo, o **Vim**) digite o seguinte código em um arquivo chamado `Vagrantfile` (o código está disponível em um repositório mas eu sugiro que inicialmente você digite para ter familiaridade com o que estamos fazendo):

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.define "testvm" do |testvm|
    testvm.vm.box = "ubuntu/trusty64"
    testvm.vm.network :private_network, ip: "192.168.33.2 "
  end

  config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--memory", "1024"]
  end

end
```

Salve e saída do seu editor predileto (`ESC :wq` no Vim) e digite *vagrant up* no terminal. Aguarde alguns segundos ou minutos de acordo com a configuração de sua máquina. Verifique quanto tempo levou para executar estes passos. Para testar, digite o comando `vagrant ssh`.

Utilize sua máquina normalmente (executar `top` para ver os processos, examinar o diretório `/etc`, instalar algum pacote manualmente).

Dentro da máquina, digite `cd /vagrant` e veja que o seu

diretório local foi mapeado como um *mount point* dentro da máquina virtual. Você pode criar este ponto com outro nome ou deixar de criá-lo de acordo com a configuração do `Vagrantfile`.

Digite `vagrant destroy` no terminal e `y` quando perguntado se pode realmente destruir esta máquina virtual.

Sem considerar a capacidade de sua máquina física e sua proficiência digitando código, vimos pelas imagens do passo a passo do VirtualBox que o número de passos para criar uma máquina virtual é grande. Some ao tempo do VirtualBox o tempo de download da imagem ISO de Ubuntu e o fato de que para apagar a máquina virtual terá que se passar por todos os passos novamente e temos um procedimento longo a ser replicado.

Após criar este `Vagrantfile`, você pode subir (*up*) a mesma máquina várias vezes, em ocasiões distintas sem utilizar a interface do VirtualBox. É possível versionar este arquivo que descreve sua máquina virtual junto com seu código e quando mudar a configuração de memória, por exemplo, esta mudança estará no histórico junto às mudanças de código.

Meça quanto tempo leva para subir uma nova máquina virtual de Ubuntu utilizando o Vagrant, agora que a imagem está gravada em sua máquina. Este cache de imagens é importante quando você está em um estágio em que precisa criar e destruir máquinas rapidamente.

Até agora, aprendemos a criar um diretório com um arquivo que descreve uma máquina virtual, a subir, conectar e destruí-la (`vagrant up`, `ssh` e `destroy`). Agora vou explicar o que este arquivo significa.

O Vagrant e a sua configuração utilizam Ruby. Existem versões distintas de APIs e vamos utilizar a versão 2. Dentro de uma

instância da configuração definimos máquinas e suas características, em um bloco de código Ruby. Veja o mesmo código comentado:

Vagrantfile

```
# -*- mode: ruby -*-
# vi: set ft=ruby :
# as linhas acima indicam a linguagem para o editor de texto

VAGRANTFILE_API_VERSION = "2"         # versão da API do Vagrant

# Este bloco de ruby contém todas as atividades que o Vagrant
# deve executar representadas pelo objeto que chamei de config
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  # config.vm cria um objeto que descreve uma máquina virtual.
  # Chamei esta vm de "testevm" e quero utilizar o Ubuntu Trusty
  # Indiquei que quero um IP privado para esta máquina
  config.vm.define "testvm" do |testvm|
    testvm.vm.box = "ubuntu/trusty64"
    testvm.vm.network :private_network, ip: "192.168.33.2 "
  end

  # Este bloco me dá acesso ao provedor da máquina virtual
  # (VirtualBox)
  # Configurei esta máquina com 1GB de memória
  config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--memory", "1024"]
  end

end
```

Neste arquivo, descrevi minha máquina virtual e disse que gostaria de usar o Ubuntu Trusty (14.04) de 64 bits. O Vagrant fornece algumas imagens e tem uma comunidade que permite aos seus usuários compartilharem imagens customizadas. É interessante notar que ele cria uma biblioteca local com as imagens que você utiliza, portanto, o custo de fazer download da imagem é amortizado entre as vezes em que você sobe e destrói suas máquinas virtuais.

O nome das imagens no ecossistema do Vagrant é *box* ou *boxes* no plural. O box que utilizei é chamado de *base box*, por ser uma

das imagens que eles mantêm só com o sistema operacional.

Além de subir e destruir sua máquina virtual, você pode desativá-la usando o comando `vagrant halt` . Leia a documentação do Vagrant sobre criar suas próprias máquinas. Neste ponto, seu exercício é explorar a máquina que criamos e customizá-la de acordo com seu gosto.

Com `vagrant up` , suba a máquina e conecte com `vagrant ssh` . Com cuidado para ter certeza de que está conectado e que não vai executar um comando em sua máquina física local, execute a sequência de comandos:

```
$ sudo apt-get update
$ sudo apt-get install nginx
```

Abra seu browser local e aponte para o endereço http://192.168.33.2. Você deve ver a mensagem de boas-vindas do *nginx*. Volte ao terminal e digite:

```
$ sudo -s
$ echo "boa cabelo" > /usr/share/nginx/html/index.html
```

Volte ao seu browser e dê reload na pagina, deve aparecer o texto *"boa cabelo"*. Imagine outras configurações que gostaria de fazer nesta máquina além de colocar um web server com um índice servindo a frase "boa cabelo". Execute e crie uma máquina personalizada para isso.

Seu próximo passo é modificar o `Vagrantfile` para utilizar esta máquina. Se errar em alguma configuração, lembre-se que é possível destruir a máquina e recriá-la do zero. Teste outros sistemas operacionais e imagens criadas para fins distintos em (https://atlas.hashicorp.com/boxes/search).

Criar a máquina rápido e ter que instalar o *nginx* manualmente é melhor do que criar usando o método manual na interface do

VirtualBox, mas ainda pode melhorar.

O Vagrant implementa o conceito de provisionador com drivers para quase todos os sistemas de gerenciamento de configuração existentes. Estes sistemas vão de receitas simples para instalar pacotes até agentes que verificam a integridade de arquivos de configuração e variáveis do sistema.

Vamos utilizar um driver de provisionamento do Vagrant que permite que um arquivo com comandos do shell seja executado logo após a criação da máquina virtual. Crie o arquivo `webserver.sh` no mesmo diretório e digite os seguintes comandos:

```
#!/bin/bash

echo "Atualizando repositórios"
sudo apt-get update
echo "Instalando o nginx"
sudo apt-get -y install nginx
```

Salve o arquivo e edite o arquivo `Vagrantfile` para ativar o provisionador. Ele deve ficar como a listagem a seguir:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.define "testvm" do |testvm|
    testvm.vm.box = "ubuntu/trusty64"
    testvm.vm.network :private_network, ip: "192.168.33.2 "
    testvm.vm.provision "shell", path: "webserver.sh"
  end

  config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--memory", "1024"]
  end

end
```

A linha `testvm.vm.provision "shell", path: webserver.sh` diz

para o Vagrant que para aquela máquina virtual deve ser usado o driver de provisionamento de *shell script* e passa como parâmetro o arquivo com comandos que criamos.

Destrua e crie a máquina virtual novamente com `vagrant destroy` e crie com `vagrant up`. Observe a saída no terminal e faça o mesmo teste no browser, sem a necessidade de se conectar por `ssh` para executar o comando que instala o *nginx*.

Vamos instalar o PHP5 na máquina. Modifique o arquivo `webserver.sh` de acordo com a listagem a seguir:

```
#!/bin/bash

echo "Atualizando repositórios"
sudo apt-get update
echo "Instalando o nginx"
sudo apt-get -y install nginx
echo "Instalando o PHP"
sudo apt-get install -y php5-fpm
```

Neste ponto, não precisamos destruir e recriar a máquina para executar novamente o provisionamento. Contamos com o fato de que o gerenciador de pacotes não instalará um pacote duas vezes. Poderíamos executar o mesmo script, que só o PHP seria instalado. O Vagrant oferece um comando que ativa somente a fase do provisionamento *vagrant provision*. Execute `vagrant provision` em seu terminal e observe a execução do script.

Até este ponto vimos como criar uma descrição de máquina virtual com o Vagrant e os seguintes comandos:

```
$ vagrant up
$ vagrant ssh
$ vagrant destroy
$ vagrant halt
$ vagrant provision
```

Com estes comandos, conseguimos gerenciar de forma rápida e limpa as máquinas virtuais criadas de acordo com o `Vagrantfile`.

Note que eles só funcionarão dentro do diretório em que o `Vagrantfile` está. Execute o comando `ls -larth` e veja que foi criado um diretório chamado `.vagrant` que contém todos os dados do ciclo de vida e provisionamento da máquina virtual.

2.5 CONCLUSÃO

Você já conhece as ferramentas de gerenciamento de máquinas virtuais instaladas em sua máquina e já consegue usá-las para tarefas básicas. Explore a documentação para entender como elas podem facilitar seu trabalho até com provedores reais de IaaS. Participe das listas de discussões e procure *Vagrantfiles* no GitHub (https://github.com) para ter novas ideias de como utilizá-lo. Explore novas imagens de sistemas operacionais.

O Vagrant abstrai muitas tarefas de gerenciamento de máquinas como automação da troca de chaves de SSH, criação de usuários e montagem de diretórios. Ele permite provisionadores plugáveis e a composição simplificada de *clusters* com máquinas de configurações distintas. O que vimos até agora do Vagrant é suficiente para continuarmos com o Ansible no próximo capítulo.

Capítulo 3

ANSIBLE

Neste capítulo, vamos retomar o exemplo do capítulo anterior usando Ansible. O Ansible é um sistema de automação de configuração feito em Python, que permite descrever procedimentos em arquivos no formato `YAML` que são reproduzidos utilizando SSH em máquinas remotas.

Existem outras ferramentas desta categoria que executam localmente e que fornecem servidores para o gerenciamento de dados remotamente, como CHEF, Puppet e CFEngine.

Vamos utilizar o Ansible localmente sem seu servidor de dados, o *Ansible Tower*. O Ansible vem com bibliotecas completas para quase todas as tarefas além de uma linguagem de template. Além das tarefas dentro de um servidor, o Ansible fornece módulos para provisionamento de máquinas e aplicações remotas. Provisionar é o jargão para instalar e configurar itens de infraestrutura e plataforma como máquinas, bancos de dados e balanceadores de carga.

No capítulo anterior, terminamos com um `Vagrantfile` e um `shell script` para instalar *nginx* e PHP. Vamos traduzir literalmente para o Ansible e progressivamente vamos utilizar seus módulos para fazer do jeito certo (*tm*). Posteriormente utilizaremos o Ansible para criar máquinas virtuais em provedores e gerenciar o ciclo de vida delas.

Poderíamos ter continuado a usar `shell script` para configurar

nossas máquinas, mas sistemas como o Ansible implementam tarefas com uma característica importante: **idempotência**. A idempotência é uma propriedade que, aplicada ao gerenciamento de configuração, garante que as operações terão o mesmo resultado independentemente do momento em que serão aplicadas. A criação de sua máquina utilizando este conjunto de configurações sempre terá o resultado previsível.

3.1 INSTALAÇÃO

Temos que instalar o Ansible em nossa máquina local. Ele não tem nenhum agente ou biblioteca que deva ser instalado na máquina virtual. O Ansible é feito em Python e funciona em muitas plataformas. A maneira como se comunica com as máquinas virtuais ou físicas configurada é por intermédio de SSH.

Se o seu sistema operacional for MacOS X, use o *homebrew*. Abra o terminal e digite `brew install ansible`. Não usa *homebrew*? Use:

```
$ sudo easy_install pip
$ sudo pip install ansible
```

Para Linux, siga a documentação em (http://docs.ansible.com/intro_installation.html#latest-releases-via-apt-ubuntu). Vou reproduzir o procedimento para Ubuntu:

```
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

Para testar se foi instalado, digite no terminal `ansible-playbook -h`. Deve aparecer a ajuda do Ansible.

3.2 A CONFIGURAÇÃO DO ANSIBLE E O

FORMATO YAML

Entre no diretório `testvm` e digite o arquivo a seguir com o nome de `webserver.yml`:

```
- hosts: all
  sudo: True
  user: vagrant
  tasks:
    - name: "Atualiza pacotes"
      shell: sudo apt-get update

    - name: "Instala o nginx"
      shell: sudo apt-get -y install nginx
```

Este é um arquivo no formato `yaml`. Use dois espaços como tab (`set ts=2` no Vim). Preste atenção no sinal de `:` (dois pontos) após os atributos. O Ansible vai reclamar se não entender o arquivo e você poderá consertá-lo.

Sempre confira a indentação e os dois pontos após os atributos. Confira também o hífen antes dos atributos quando necessário. A formatação é simples mas importante.

`YAML` é um formato que serve para serialização e transmissão de dados. Ele é composto de palavras-chaves separadas de um valor por `:` (dois pontos). Seus dados podem ser chaves e valores, listas de chaves e valores e dicionários. Por exemplo:

Uma lista:

```
- gato
- cachorro
- limão
```

Dicionário:

```
nome: gleicon
sobrenome: moraes
linux: sim por favor
```

Os valores podem ser alfanuméricos. A combinação segue o

padrão de `tabspace=2` (cada nível é separado por 2 espaços). É um formato simples e o Ansible utiliza um subset de tudo que ele pode representar. Você pode usar o YAML Lint em (http://www.yamllint.com/) para verificar seu playbook sem ter que executá-lo.

Vamos ver mais sobre o arquivo de configuração neste e nos próximos capítulos. O nome correto deste arquivo é `playbook`, pois o Ansible tem um arquivo de configuração de suas propriedades que não dependem de um host específico.

3.3 ANSIBLE E VAGRANT

Abra o arquivo `Vagrantfile` e modifique-o para que fique de acordo com a listagem a seguir:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.define "testvm" do |testvm|
    testvm.vm.box = "ubuntu/trusty64"
    testvm.vm.network :private_network, ip: "192.168.33.2 "

    testvm.vm.provision "ansible" do |ansible|
      ansible.playbook = "webserver.yml"
      ansible.verbose = "vvv"
    end

  end

  config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--memory", "1024"]
  end

end
```

A diferença entre os dois arquivos é o bloco a seguir:

```
testvm.vm.provision "ansible" do |ansible|
  ansible.playbook = "webserver.yml"
  ansible.verbose = "vvv"
end
```

A definição de máquina virtual de `testvm` tem um atributo que define um provisionador. No capítulo anterior, usei um `shell script` para instalar o *nginx*. Com esta mudança, o Vagrant utilizará o Ansible como provisionador com o playbook `webserver.yml`. Deixei o atributo `verbose` do Ansible com o valor `vvv` para indicar que quero todas as mensagens enviadas para o console.

Vamos examinar o arquivo `webserver.yml` linha a linha e adicionar a instalação do PHP. A primeira linha que contém o atributo `hosts` é o nível mais alto do playbook. Ela define em qual maquina ou grupo de máquinas a descrição será aplicada. O valor `all` indica que pode ser aplicado a qualquer uma definida no arquivo que contém os endereços das máquinas. Como o Vagrant executa o Ansible indiretamente, ele fornece este arquivo.

Se executarmos o playbook diretamente no terminal sem o Vagrant, teríamos que passar um arquivo de inventário (em inglês, `inventory file` ou `hosts.ini`). O formato deste arquivo é simples:

```
[grupo1]
10.0.0.1
10.0.0.2
10.0.0.3

[grupo2]
10.0.0.4
10.0.0.5
```

Se o valor do atributo `hosts` fosse `grupo1` e estivéssemos executando o Ansible manualmente:

```
$ ansible-playbook -i hosts_file webserver.yml
```

O playbook seria aplicado apenas nas máquinas com IPs 10.0.0.1, 10.0.0.2, 10.0.0.3. Se o valor fosse `all`, ele seria aplicado

em todas as máquinas. Este é o caso utilizado caso você tenha uma máquina virtual em algum provedor e queira executar o playbook.

Basta se certificar de que seu usuário tem autenticação por chave de SSH (`ssh usuario@maquina` deve logar sem pedir senha). Ajuste o nome do usuário no arquivo do Ansible e execute o playbook como indicado anteriormente.

Você não precisará se preocupar com este arquivo agora, mas posteriormente vamos utilizar o mesmo playbook para desenvolvimento local e deploy em uma VPS e este arquivo será util.

```
sudo: True
user: vagrant
```

Estes atributos indicam que o Ansible pode usar *sudo* e o usuário de conexão na máquina é o usuário `vagrant` . Este parâmetro também será modificado para instalação em uma VPS para seu usuário ou o usuário default da distribuição.

```
tasks:
  - name: "Atualiza pacotes"
    shell: sudo apt-get update

  - name: "Instala o nginx"
    shell: sudo apt-get -y install nginx
```

O atributo `tasks` é uma lista de tarefas que o Ansible vai executar. Ela é composta por nome/ação. Neste arquivo, utilizamos uma task chamada `shell` que executa o comando em seguida diretamente no terminal da máquina.

Agora, execute `vagrant up` e observe a instalação do Ansible até receber um status final semelhante a este:

```
PLAY RECAP *********************************************************
testvm       : ok=3    changed=2    unreachable=0    failed=0
```

`PLAY RECAP` é o resumo da aplicação do playbook. Para o host `testvm` duas tasks (`ok=3`) foram executadas. Uma delas causou

uma mudança no host (`changed=2`). Durante a execução do provisionamento cada uma das tasks foi executada e indicada pelo Ansible, além do passo chamado *GATHERING FACTS*.

Como nosso `Vagrantfile` tem um indicador de nível de verbosidade do Ansible configurado, você viu bem mais do que isso. Você pode remover esta opção quando terminar um playbook pois ela só é útil para debug durante desenvolvimento.

Se executarmos o provisionamento novamente na mesma máquina com `vagrant provision` , teremos a mesma saída:

```
PLAY RECAP *********************************************************
testvm       : ok=3    changed=2    unreachable=0    failed=0
```

O Ansible executou a mesma sequência de comandos nas duas vezes. A task `shell` é uma das mais simples do Ansible e tem pouco controle de estado, o que não contribui com a idempotência entre as rodadas do provisionamento. Nesta configuração, o controle é executar as duas tasks com sucesso sem guardar um *estado* atual ou levá-lo em conta. Vamos melhorar isso e aprender como utilizar melhor os módulos que o Ansible oferece.

3.4 REFATORAÇÃO

Toda task é uma função de um módulo do Ansible. Os módulos que vêm na instalação padrão ele são chamados de *Core Modules*. Você pode construir seus módulos utilizando a linguagem Python e estender o Ansible.

Vamos refatorar nosso playbook para utilizar um módulo core chamado `apt_module` (http://docs.ansible.com/apt_module.html) em vez do módulo `shell` . O módulo `shell` é útil para automatizar muitas tarefas, mas você vai encontrar módulos especializados do Ansible que cuidam da consistência da tarefa e criam estados para manter a idempotência. Atualize o arquivo `webserver.yml` de

acordo com a listagem a seguir.

```
- hosts: all
  sudo: True
  user: vagrant
  tasks:
    - name: "Atualiza pacotes e instala nginx"
      apt: name=nginx state=latest update_cache=yes
      install_recommends=yes
```

Reduzimos as duas tasks para uma que utiliza o módulo `apt` e diz para o Ansible instalar o *nginx* no último release presente no repositório. Um dos atributos desta task é indicar que um update no cache do gerenciador de pacotes deve executado.

Outro atributo que utilizaremos é a indicação de que as dependências recomendadas devem ser instaladas automaticamente. Note que, utilizando o módulo `shell`, eu precisei pedir explicitamente pela atualização do repositório e também adicionar o parâmetro `-y` ao comando `apt-get install` para evitar que a task ficasse esperando uma entrada do usuário.

O Ansible também fornece um módulo para o gerenciador de pacotes yum (http://docs.ansible.com/yum_module.html) e diretivas condicionais que podem ser usadas para detectar o sistema operacional e distribuições de Linux. Veja como declarar a mesma task para instalar *nginx* para duas famílias de distribuições de Linux utilizando o condicional `when` e variáveis internas do Ansible:

```
- name: Install the nginx packages
  yum: name={{ item }} state=present
  with_items: redhat_pkg
  when: ansible_os_family == "RedHat"

- name: Install the nginx packages
  apt: name={{ item }} state=present update_cache=yes
  with_items: ubuntu_pkg
  environment: env
  when: ansible_os_family == "Debian"
```

Podemos testar o novo `webserver.yml` com `vagrant up` ou

apenas `vagrant provision` caso sua máquina virtual ainda esteja sendo executada. Após a primeira execução do `vagrant up` ou `vagrant provision`, execute `vagrant provision` novamente e analise a saída no console. O *PLAY RECAP* deve ser diferente.

Execute o provisionamento em seguida para notar a diferença entre a primeira e a segunda execução. Esta é uma maneira simples de verificar a idempotência do playbook: na segunda rodada, o atributo `changed` deve ser 0.

Após `vagrant up`:

```
PLAY RECAP *********************************************************
testvm       : ok=2    changed=1    unreachable=0    failed=0
```

Após `vagrant provision`:

```
PLAY RECAP *********************************************************
testvm       : ok=2    changed=0    unreachable=0    failed=0
```

Este módulo nos deu mais controle e um estado: a interpretação do gerenciador de pacotes. Existem módulos para RPM e outras distribuições de Linux além da possibilidade de automatizar a instalação ou geração de um pacote a partir de um repositório de código.

O parâmetro `state` configurado para `latest` descreve dois estados: o pacote deve estar instalado e deve ser a última versão disponível. Se no momento da execução estas condições forem verdadeiras, nenhuma ação será tomada. Se alguma delas for falsa, a ação correta será executada utilizando o gerenciador de pacotes `apt`.

3.5 CONCLUSÃO

Neste capítulo, vimos como o Ansible funciona e como sair de um provisionamento baseado em `shell script` para o

provisionamento baseado no Ansible.

Explore a documentação e módulos do Ansible para entender como a biblioteca que vem com ele aborda tarefas de administração de sistemas como instalação de pacotes, arquivos de configuração e diferentes distribuições de sistemas operacionais.

Procure mais informações sobre outros sistemas de gerenciamento de configuração já citados: CFEngine (http://cfengine.com/), Puppet (https://puppetlabs.com/) e CHEF (https://www.chef.io/chef/). Entenda como a arquitetura deles interfere na sua arquitetura de sistemas atual e qual o comprometimento de tempo e estrutura está envolvido na adoção de um sistema mais completo e complexo.

CAPÍTULO 4

INSTALANDO WORDPRESS EM UMA MÁQUINA

Neste capítulo vamos montar um playbook que instale e configure um WordPress completo, utilizando o *nginx* como servidor HTTP, o PHP-FPM e MySQL. Existem playbooks completos para esta tarefa, mas vamos montar o nosso com o objetivo simples de ter um blog funcionando dentro de nossa máquina virtual.

Poderia ter escolhido Apache, mas é uma oportunidade interessante de usar PHP como utilizamos outros interpretadores, fora do web server. A sequência de tarefas apresentada é parte de um *framework* simples para o deploy de aplicações direto do código-fonte. Você verá posteriormente que se encararmos a aplicação desta forma, o deploy não é tão diferente entre as linguagens.

4.1 EXPANDINDO NOSSO PLAYBOOK — UM BLOG COM WORDPRESS

Vamos pensar nas tarefas para sair de uma máquina com Linux nova para um blog:

- Criar um novo projeto;
- Criar um Vagrantfile que use o Ansible para provisionamento;
- Instalar *nginx*;

- Instalar PHP (PHP-FPM);
- Instalar MySQL;
- Configurar o *nginx* para interpretar arquivos PHP com o PHP-FPM;
- Fazer download e instalar o WordPress;
- Configurar um site no *nginx* que execute o WordPress.

O código para este projeto está no repositório do livro, mas vou descrever passo a passo como estruturá-lo. Vamos criar um diretório chamado `blogvm`, criar um `Vagrantfile` e um arquivo chamado `blog.yml` de acordo com as listagens a seguir:

Vagrantfile:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.define "blogvm" do |blogvm|
    blogvm.vm.box = "ubuntu/trusty64"
    blogvm.vm.network :private_network, ip: "192.168.33.2 "

    blogvm.vm.provision "ansible" do |ansible|
      ansible.playbook = "blog.yml"
      ansible.verbose = "vvv"
    end

  end

  config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--memory", "1024"]
  end

end
```

blog.yml:

```
- hosts: all
  sudo: True
  user: vagrant
```

```
  tasks:
    - name: "Atualiza pacotes e instala nginx"
      apt: name=nginx state=latest update_cache=yes
      install_recommends=yes
```

Estes arquivos são as versões finais do capítulo anterior. Seguindo a nossa lista, coloquei a instalação de PHP-FPM e MySQL. Uma das maneiras de utilizar o PHP com *nginx* é utilizar a versão FPM do PHP. É uma implementação alternativa do protocolo FastCGI que já tem um gerenciador de processos. A interface entre o servidor HTTP e o interpretador será feita por um *Unix Domain Socket* que o PHP-FPM disponibiliza.

Nossa próxima tarefa é criar um arquivo de configuração no *nginx* para um site que saiba intepretar arquivos com extensão PHP. Vou explicar como isso fica no *nginx* e posteriormente como faremos o Ansible montar o arquivo de configuração correto. Para facilitar, vou criar o `document root` no lugar onde o WordPress ficará instalado, `/opt/wordpress`.

```
server {
        listen   80;
        server_name  blog localhost;

        access_log  /var/log/nginx/blog.access.log;

        root   /opt/wordpress;
        index  index.html index.htm index.php;

        location / {
                try_files $uri $uri/ /index.html;
        }

        location ~ \.php$ {
                try_files $uri =404;
                fastcgi_pass unix:/var/run/php5-fpm.sock;
                fastcgi_index index.php;
                fastcgi_param SCRIPT_FILENAME
                $document_root$fastcgi_script_name;
                include fastcgi_params;
        }
}
```

O *nginx* pode servir vários sites. A definição de cada um dele fica em uma diretiva chamada *server*. No Ubuntu, a estrutura de configuração fica em `/etc/nginx` e os sites dentro de `/etc/nginx/sites-available` com um link dentro de `/etc/nginx/sites-enabled` quando ativo.

A próxima tarefa é transportar a configuração do nosso site do *nginx* para esta estrutura. Vamos criar três tasks: criar o `*/opt/wordpress` , copiar a configuração e ativar o novo site. Para facilitar, vamos remover o site default e reiniciar o *nginx*.

Para testar estes passos, vamos colocar uma task que vai criar um arquivo simples chamado `test.php` com o comando `php_info()` . Nosso teste é abrir o IP 192.168.33.2 owser e ver a tela de informações sobre o PHP impressa.

```
<?php phpinfo(); ?>
```

```
- hosts: all
  sudo: True
  user: vagrant
  tasks:
    - name: "Atualiza pacotes e instala nginx"
      apt: name=nginx state=latest update_cache=yes
      install_recommends=yes

    - name: "Instala PHP-FPM"
      apt: name=php5-fpm state=latest install_recommends=yes

    - name: "Instala MySQL"
      apt: name=mysql-server state=latest install_recommends=yes

    - name: "Cria diretório /opt/wordpress"
      shell: mkdir -p /opt/wordpress

    - name: "Copia configuração de blog.nginx para
      /etc/nginx/sites-available/blog"
      copy: src=blog.nginx dest=/etc/nginx/sites-available/blog

    - name: "Ativa o site"
      shell: ln -fs /etc/nginx/sites-available/blog
      /etc/nginx/sites-enabled/blog
```

```
- name: "Apaga o site default"
  shell: rm -f /etc/nginx/sites-enabled/default

- name: "Reinicia o NGINX"
  shell: service nginx restart

- name: "Cria uma pagina de teste do PHP"
  copy: src=test.php dest=/opt/wordpress
```

Neste playbook, introduzi uma nova task, chamada `copy` que copia arquivos locais para o host. Também voltei a utilizar a task `shell` como o modo mais simples de executar uma tarefa.

Vamos executar o fluxo completo. Se sua máquina virtual está criada, use `vagrant destroy` para apagá-la e `vagrant up` para executar nossa instalação. Abra seu browser e aponte para http://192.168.33.2/test.php.

PHP Version 5.5.9-1ubuntu4.6

System	Linux vagrant-ubuntu-trusty-64 3.13.0-32-generic #57-Ubuntu SMP Tue Jul 15 03:51:08 UTC 2014 x86_64
Build Date	Feb 13 2015 19:15:42
Server API	FPM/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/etc/php5/fpm
Loaded Configuration File	/etc/php5/fpm/php.ini
Scan this dir for additional .ini files	/etc/php5/fpm/conf.d
Additional .ini files parsed	/etc/php5/fpm/conf.d/05-opcache.ini, /etc/php5/fpm/conf.d/10-pdo.ini
PHP API	20121113
PHP Extension	20121212
Zend Extension	220121212
Zend Extension Build	API220121212,NTS
PHP Extension Build	API20121212,NTS
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring
IPv6 Support	enabled
DTrace Support	enabled
Registered PHP Streams	https, ftps, compress.zlib, compress.bzip2, php, file, glob, data, http, ftp, phar, zip
Registered Stream Socket Transports	tcp, udp, unix, udg, ssl, sslv3, tls
Registered Stream Filters	zlib.*, bzip2.*, convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, dechunk

This program makes use of the Zend Scripting Language Engine:
Zend Engine v2.5.0, Copyright (c) 1998-2014 Zend Technologies
with Zend OPcache v7.0.3, Copyright (c) 1999-2014, by Zend Technologies

Powered By Zend Engine 2

Figura 4.1: test.php

PLAY RECAP

```
PLAY RECAP *********************************************************
blogvm       : ok=10   changed=9    unreachable=0    failed=0
```

Se o teste falhar ou o seu *PLAY RECAP* estiver diferente, observe a saída do provisionamento por erros. A saída do `test.php` pode ser um pouco diferente dependendo de atualizações de bibliotecas ou componentes do sistema operacional.

Vamos colocar o WordPress neste playbook e configurá-lo com o MySQL como banco de dados. As distribuições de Linux vêm com

pacotes do WordPress, mas vamos baixar a versão **latest** do site para ilustrar como instalar um software diretamente do código-fonte. Se você navegar até (https://wordpress.org) na sessão de downloads verá que o último release sempre tem o nome de *latest*.

A URL https://wordpress.org/latest.tar.gz é a que utilizaremos para fazer o download do WordPress. Também vamos utilizar módulos do Ansible em substituição às tarefas de criar um link simbólico e apagar um diretório.

Introduzi novas diretivas, como `unarchive`, `url_get`, módulos do MySQL e novos usos do módulo `file`. Temos o conceito de variáveis `vars` e de templates.

Vamos experimentar o playbook antes da explicação completa. Se você estiver executando diretamente do repositório, pode pular para depois da listagem, pois vamos ver algumas tasks especiais e discutir como refatorar este playbook.

A listagem a seguir instala o Wordpress completo em uma máquina virtual:

```
- hosts: all
  sudo: True
  user: vagrant
  vars:
    mysql_root_password: root
    mysql_wp_user: wordpress
    mysql_wp_password: wordpress
    wordpress_db_name: wordpress

  tasks:
    - name: "Atualiza pacotes e instala nginx"
      apt: name=nginx state=latest update_cache=yes
           install_recommends=yes

    - name: "Instala PHP-FPM"
      apt: name=php5-fpm state=latest install_recommends=yes

    - name: "Instala MySQL"
      apt: name=mysql-server state=latest install_recommends=yes
```

```
- name: "Instala Extensões de MySQL para PHP"
  apt: name=php5-mysql state=latest install_recommends=yes

- name: "Instala biblioteca python-mysqldb"
  apt: name=python-mysqldb state=latest install_recommends=yes

- name: "Copia configuração de blog.nginx para
        /etc/nginx/sites-available/blog"
  copy: src=blog.nginx dest=/etc/nginx/sites-available/blog

- name: "Apaga o site default"
  file: path=/etc/nginx/sites-enabled/default state=absent

- name: "Ativa o site"
  file: src=/etc/nginx/sites-available/blog
        dest=/etc/nginx/sites-enabled/blog state=link

- name: "Reinicia NGINX"
  service: name=nginx state=restarted

- name: "Cria /opt/wordpress"
  file: dest=/opt/wordpress mode=755
        state=directory owner=www-data

- name: "Download do Wordpress"
  get_url: url=https://wordpress.org/latest.tar.gz
           dest=/tmp/latest.tar.gz

- name: "Abre wordpress em /opt/wordpress"
  unarchive: src=/tmp/latest.tar.gz dest=/opt copy=no

- name: "Corrige permissões"
  file: path=/opt/wordpress recurse=yes owner=www-data
        group=www-data

- name: "Inicia MySQL"
  service: name=mysql state=started enabled=true

- name: "Cria .my.cnf"
  template: src=my.cnf.j2 dest=~/.my.cnf mode=0600

- name: "Cria senha de root para root@mysql"
  mysql_user: name=root
          password="{{ mysql_root_password }}"
          check_implicit_admin=yes
          priv="*.*:ALL,GRANT"
          state=present
```

```
            host="{{ item }}"
  with_items:
    - "{{ ansible_hostname }}"
    - 127.0.0.1
    - *1
    - localhost

- name: "Cria wordpress database"
  mysql_db: name=wordpress
            login_user=root
            login_password="{{ mysql_root_password }}"
            state=present

- name: "Cria usuário wordpress"
  mysql_user: name="{{ mysql_wp_user }}"
            password="{{ mysql_wp_password}}"
            priv="{{ wordpress_db_name }}".*:ALL
            check_implicit_admin=yes
            login_user=root
            login_password="{{ mysql_root_password }}"
            host="{{ item }}"
            state=present
  with_items:
    - "{{ ansible_hostname }}"
    - 127.0.0.1
    - *1
    - localhost
```

Aponte seu browser para http://192.168.33.2 e configure o WordPress. Estes passos também podem ser automatizados mas fica como exercício.

Este playbook contém uma sessão nova chamada `vars` . Estas variáveis podem ser usadas no próprio playbook, como nas tasks de MySQL ou em arquivos externos chamados templates. A sintaxe é simples baseada no engine de templates *Jinja2*.

De todos os recursos do Jinja2 só utilizaremos a substituição de variáveis entre chaves duplas, mas a linguagem possui desvios condicionais, loops e muitos recursos para criar templates inteligentes. Eu recomendo ler a documentação, mas também manter os templates simples: template module (http://docs.ansible.com/template_module.html).

Algumas tasks têm uma construção semelhante ao exemplo a seguir:

```
- name: "Cria usuário wordpress"
      mysql_user: name="{{ mysql_wp_user }}"
              password="{{ mysql_wp_password}}"
              priv="{{ wordpress_db_name }}".*:ALL
              check_implicit_admin=yes
              login_user=root
              login_password="{{ mysql_root_password }}"
              state=present
              host="{{ item }}"
      with_items:
        - "{{ ansible_hostname }}"
        - 127.0.0.1
        - *1
        - localhost
```

Esta estrutura permite que passemos uma lista de itens à qual a task será aplicada. Estes itens podem ser locais com `with_items` ou provenientes de um template ou variável. São úteis para manter o código modular sem repetições desnecessárias. Vale a regra de manter o playbook simples sem utilizar estruturas complexas para o loop das tasks. Neste caso, os itens na sessão *with_items* são hosts utilizados para a criação do usuário. Os direitos (*GRANTS*) dos usuários do MySQL são específicos para cada host ou IP de origem da conexão (o mesmo usuário pode ter permissão de criar tabelas a partir do *localhost*, mas só ler, se vindo de outro host).

As tasks de MySQL deste playbook foram estruturadas assumindo que a instalação do pacote do `mysql_server` cria um usuário root sem senha.

Para que o template tenha comportamento idempotente em todas as execuções, seguimos a recomendação da documentação do Ansible: instalamos um módulo de Python para MySQL, criamos uma senha para o usuário root e um arquivo chamado `.my.cnf` que mora no diretório `/root`. Existem outras maneiras de trabalhar com autenticação e tokens que são mais seguras, que você pode

pesquisar na documentação do Ansible.

O playbook funciona e poderíamos parar aí para instalar um blog. Tudo está na mesma máquina, você pode subir e descer quantas vezes quiser, modificar parâmetros até ficar a contento e posteriormente fazer o deploy em uma máquina de produção.

REFATORAÇÃO

Vamos refatorar este playbook para introduzir a ideia de *roles*. Um playbook tem roles (papéis em português) que agem sobre partes específicas do sistema operacional. Esta mudança contribui para a modularidade e reaproveitamento de código: um role pode ser utilizado em vários playbooks. A refatoração contribui também para a legibilidade do playbook.

Vamos separar nosso playbook em 5 roles: `common` (sistema operacional e tarefas genéricas), `nginx`, `php`, `mysql` e `wordpress`. Vamos separar também as variáveis e criar um modelo de variáveis para evitar que coloquemos senhas e tokens de acesso por engano em um repositório de código. A mudança não introduz tarefas diferentes, apenas muda a organização dos arquivos. A árvore de diretórios ficará assim:

```
|-- .gitignore
|-- blog.yml
|-- roles
|   |-- common
|   |   |-- tasks
|   |   |   |-- main.yml
|   |   |-- templates
|   |-- mysql
|   |   |-- tasks
|   |   |   |-- main.yml
|   |   |-- templates
|   |   |   |-- my.cnf.j2
|   |-- nginx
|   |   |-- tasks
|   |   |   |-- main.yml
```

```
|   |   |-- templates
|   |   |   |-- blog.nginx
|   |-- php
|   |   |-- tasks
|   |   |   |-- main.yml
|   |   |-- templates
|   |-- wordpress
|   |   |-- tasks
|   |   |   |-- main.yml
|   |   |-- templates
|-- Vagrantfile
|-- vars
|   |-- mysql.yml
|   |-- mysql.yml.dist
```

Agora temos um diretório chamado `roles` com cinco subdiretórios, cada um representando um elemento que descrevemos anteriormente. Dentro de cada diretório de role temos os subdiretórios `task` e `templates`. As tasks moram dentro do arquivo `main.yml`.

Você pode dividir em quantos arquivos quiser para organizar, mas neste modelo usaremos apenas o `main.yml`. Dentro do subdiretório `templates` ficam os arquivos `.j2` ou arquivos que vamos copiar para o servidor.

O diretório `vars` na raiz do projeto tem arquivos com variáveis que serão utilizadas pelos roles. Eu criei o `mysql.yml.dist` que é o modelo do arquivo. No `.gitignore` coloquei propositalmente o caminho `vars/mysql.yml` para evitar que minhas credenciais sejam versionadas. Você vai copiar o `mysql.yml.dist` para `mysql.yml` dentro do mesmo diretório e completá-lo com suas credenciais e senhas.

O arquivo `blog.yml` ficou bem simples e agora utiliza a diretiva `roles:`

```
- hosts: all
  sudo: True
  user: vagrant
  vars_files:
```

```
    - vars/mysql.yml
  roles:
    - common
    - mysql
    - php
    - nginx
    - wordpress
```

O exercício deste capítulo é mais uma refatoração. Existe uma condição que faz com que o role `wordpress` dependa do role `common` : a criação do diretório `/opt/wordpress` .

Você poderia utilizar o role `wordpress` em outro playbook mas precisaria criar este diretório.

A melhor forma de remover esta dependência é criar uma variável para o diretório destino, substituir todas as ocorrências de "/opt/wordpress" por ela e em cada role criar o diretório caso ele não exista.

CONCLUSÃO

Neste capítulo, vimos novas tasks do Ansible e um exemplo completo de como fazer o deploy de uma aplicação em uma máquina virtual. Se você imaginar que no lugar do WordPress você acessou o repositório do seu código, pode ver que não é difícil criar um ambiente portável para desenvolvimento e a estrutura para deploy com o que vimos até aqui.

É interessante notar que a progressão do desenvolvimento do playbook é semelhante ao processo que usamos para programar.

Até refatoração fizemos e, como os arquivos estão estruturados, poderíamos ter versionado com Git ou outro controle de versão. Você poderia ter versões de software ou máquinas sendo executadas para testes de regressão ou performance rapidamente.

A última refatoração que fizemos organizou o playbook em roles que podem ser reaproveitados em outros playbooks, assim como os componentes que o Ansible provê como módulos e playbooks do Ansible Galaxy (http://galaxy.ansible.com).

Capítulo 5

PROXY REVERSO E WORDPRESS EM DUAS MÁQUINAS

Este capítulo é ousado mas simples: vamos aproveitar o que vimos anteriormente para melhorar nosso playbook e usar o Vagrant para criar uma estrutura que tem mais de uma máquina. Antes vamos entender um pouco mais a técnica de proxy reverso com o *nginx* e como este padrão de arquitetura pode ser usado para diferentes interpretadores e linguagens.

5.1 NGINX E PROXY REVERSO

O exemplo deste capítulo e do anterior servem para outras aplicações que utilizam PHP e MySQL. O que precisaria ser modificado para instalar um CMS (*Content Management System*) como o Joomla (http://www.joomla.org/)? E para outras linguagens e plataformas ?

Nestes casos, devemos voltar a atenção ao passo de configuração: se depende de um arquivo com os dados das máquinas instaladas, se precisa de passos completados pelo browser ou se podemos fornecer um arquivo com as configurações corretas, sem intervenção manual.

A maneira como configuramos o *nginx* é chamada de *proxy*

reverso. Se em vez de executar o PHP-FPM uma aplicação em Python estivesse ouvindo em uma porta local, não mudaria muito o playbook.

Vamos construir uma aplicação simples que ouve na porta 10000 e responde uma API de *echo*: ela recebe um `POST` com o parâmetro `msg` e imprime o valor de volta. Se o método chamado for `GET`, um formulário com uma caixa de texto aparecerá.

É um exemplo simples mas demonstra o conceito de proxy reverso e de outro interpretador. Este exemplo está no diretório `python-nginx` do repositório com o nome de `http_server.py`:

```
#!/usr/bin/python

import SimpleHTTPServer
import SocketServer
import cgi
import socket

PORT=10000

class ReuseAddrTCPServer(SocketServer.TCPServer):
    def server_bind(self):
        self.socket.setsockopt(socket.SOL_SOCKET,
                               socket.SO_REUSEADDR, 1)
        self.socket.bind(self.server_address)

class ServerHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
    def do_GET(self):

        self.wfile.write("<html><body><form action='/'
                         method=POST>")
        self.wfile.write("<textarea name=msg rows='10'
                         cols='100'></textarea>")
        self.wfile.write("<br><input type='submit'
                         name='submit'>")
        self.wfile.write("</form></body></html>")
        return

    def do_POST(self):
        self.send_response(200)
        self.end_headers()
        form = cgi.FieldStorage(
```

```
            fp=self.rfile,
            headers=self.headers,
            environ={'REQUEST_METHOD':'POST',
                     'CONTENT_TYPE':self.headers['Content-Type'],
                     })

        for field in form.keys():
            self.wfile.write('\t%s=%s\n' % (field,
                                            form[field].value))

        return

ReuseAddrTCPServer(("", PORT), ServerHandler).serve_forever()
```

Este código cria um servidor HTTP com as bibliotecas padrão do Python, ouve na porta 10000 e só responde a requests HTTP do tipo `GET` e `POST`. Para testá-lo, abra dois terminais; em um deles execute o comando:

```
$ python http_server.py
```

No outro, o comando `curl` para mandar um request `POST` para nosso servidor:

```
$ curl -X POST -d "msg=aaaa&oi=123" http://localhost:10000
```

No terminal em que o servidor está sendo executado, a saída deve ser parecida com:

```
$ curl -X POST -d "msg=aaaa&oi=123" http://localhost:10000
msg=aaaa
oi=123
$
```

O servidor não tem o mínimo necessário para uma boa aplicação web como logs, monitoração, arquivo de configuração para dados como a porta e o endereço, mas servirá para o propósito de demonstrar a mudança em nosso playbook para utilizar uma aplicação Python e o *nginx* como proxy reverso.

O `Vagrantfile` deste exemplo é parecido com o do capítulo anterior: apenas uma máquina virtual será criada e o arquivo

`web.yml` será executado. Dentro desta máquina, teremos o *nginx*, Python e nosso programa. Ele será executado dentro de um programa chamado *supervisord*.

O supervisord é um programa que cuida da execução de outros programas: redireciona a saída padrão para um arquivo de log e monitora sua execução para reiniciar caso o script pare de responder.

Dentro desta máquina, nosso `http_server.py` será executado e vai ouvir na porta 10000 de `localhost`.

O *nginx* terá um site que vai ouvir no IP 192.168.33.10 e que fará o request para este serviço de *backend*. Ele vai gerenciar as conexões vindas de usuários e as respostas vindas do backend.

Este é o `web.yml`:

```
- hosts: all
  sudo: True
  user: vagrant
  vars_files:
    - vars/mysql.yml
  roles:
    - common
    - nginx
    - app
```

Para organizar o deploy da aplicação, criei um role chamado `app`. Este role cria um diretório `/opt/app`, copia o `http_server.py` para lá, configura o `supervisord` e inicia a aplicação. Este é o arquivo `tasks/main.yml` do role *app*

```
- name: "Cria o diretório /opt/app se ele não existe"
  file: dest=/opt/app mode=755 state=directory owner=www-data

- name: "Copia o arquivo http_server.py para /opt/app"
  copy: src=http_server.py dest=/opt/app/

- name: "Cria a entrada app para o supervisord"
  template: src=app.conf dest=/etc/supervisor/conf.d/
```

```
- name: "Reinicia supervisord"
  service: name=supervisor state=restarted

- name: "Inicia a aplicação"
  supervisorctl: name=app state=started
```

No role `common`, instalamos o pacote `supervisord` para suportar a task `supervisorctl` e o controle da aplicação. Nos passos anteriores implementamos o que já discutimos.

A configuração do *nginx* mudou um pouco:

```
server {
    listen   80;
    server_name  app localhost;
    access_log  /var/log/nginx/app.access.log;
    root   /opt/app;
    index  index.html index.htm index.php;

    location / {
    proxy_pass                  http://127.0.0.1:10000/;
    proxy_redirect              off;
    proxy_set_header            Host            $host;
    proxy_set_header            X-Real-IP       $remote_addr;
    proxy_set_header            X-Forwarded-For
                                $proxy_add_x_forwarded_for;

    client_max_body_size        5M;
    client_body_buffer_size     5M;

    proxy_connect_timeout       30;
    proxy_send_timeout          30;
    proxy_read_timeout          30;

    proxy_buffer_size           16k;
    proxy_buffers               16 128k;
    proxy_busy_buffers_size     512k;
    proxy_temp_file_write_size  512k;
        }
}
```

Suba a máquina com `vagrant up` e acesse http://192/168.33.10 em seu browser. Digite algo e clique em `submit`. Repita os testes com o `curl` no terminal:

```
$ curl -vvv -X POST -d "msg=oie" http://192.168.33.10 /
```

Entre na máquina virtual com `vagrant ssh` e teste o serviço localmente:

```
$ curl -vvv -X POST -d "msg=oie" http://localhost:10000/
```

O resultado deve ser semelhante, mas com os headers da biblioteca *SimpleHTTPServer* do Python. Neste teste, você acessou o backend diretamente, antes do *nginx*. Analise os logs em `/var/log/nginx/` e o log da aplicação em `/var/log/app.log`, redirecionado pelo supervisord.

Não é o objetivo fazer uma análise das opções de configuração do *nginx*, basta dizer que a diretiva `proxy_pass` redireciona o trafego que chega na URI `/` para o backend.

As opções em seguida são relacionadas a *HTTP HEADERS* que devem ser repassados para a aplicação de backend e o tamanho dos buffers disponíveis. Com o *nginx* é possível fazer proxy, *load balancing* e cache além de servir páginas HTTP. Procure mais informações em (http://nginx.org/en/docs/).

O motivo deste desvio foi técnica que utilizamos para *fastcgi* e *HTTP* em aplicações de backend.

5.2 WORDPRESS EM DUAS MÁQUINAS

O objetivo agora é simples: quero que meu banco de dados fique em uma máquina e o WordPress em outra. Quero rodar este ambiente localmente com o Vagrant, mas quero que o playbook esteja pronto para provisionar em duas máquinas externas.

Vamos usar o `Vagrantfile` para criar as máquinas e aplicar o provisionamento correto em cada uma. Dividiremos nossos roles para indicar o que pertence ao banco de dados e o que pertence ao webserver com WordPress. A documentação do Vagrant em (http://docs.vagrantup.com/v2/multi-machine/) sobre isso é bem

clara. Adaptando ao nosso `Vagrantfile` , fica assim:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"


Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|

  config.vm.define "web" do |webvm|
    webvm.vm.box = "ubuntu/trusty64"
    webvm.vm.network :private_network, ip: "192.168.33.2 "

    webvm.vm.provision "ansible" do |ansible|
          ansible.playbook = "blog.yml"
      ansible.verbose = "vvv"
      ansible.groups = {
        "web" => ["192.168.33.2 "],
        "db" =>  ["192.168.33.2 "],
          }
      ansible.limit = "web"
    end
  end

  config.vm.define "db" do |dbvm|
    dbvm.vm.box = "ubuntu/trusty64"
    dbvm.vm.network :private_network, ip: "192.168.33.2 "

    dbvm.vm.provision "ansible" do |ansible|
          ansible.playbook = "blog.yml"
      ansible.verbose = "vvv"
      ansible.groups = {
        "web" => ["192.168.33.2 "],
        "db" =>  ["192.168.33.2 "],
     }
      ansible.limit = "db"
    end
  end

  config.vm.provider "virtualbox" do |v|
    v.customize ["modifyvm", :id, "--memory", "1024"]
  end

end
```

As mudanças no `Vagrantfile` foram simples: duas sessões para

criar máquinas virtuais distintas. Como os componentes desta arquitetura são independentes, uma máquina web e uma de banco de dados, criei dois grupos e programaticamente usei o Vagrant para indicar ao Ansible o que cada uma representa. No momento de configurar o WordPress pela interface web, coloque o IP 192.168.33.2 no campo *hostname*. É a única mudança em relação ao setup anterior.

Após a configuração e o primeiro post no WordPress, use `vagrant ssh db` e `vagrant ssh web` para inspecionar as duas máquinas. Em um setup com mais de uma máquina o, Vagrant usa o nome que demos para `config.define` como identificador para seus comandos. Para provisionar novamente só a máquina de banco, `vagrant provision db` ; para se conectar à máquina de frontend `vagrant ssh web` ; o mesmo para `destroy` , `halt` e `up` .

Poderíamos automatizar totalmente a configuração do WordPress sem utilizar a interface web, com templates de arquivos no formato correto com as variáveis. É uma tarefa relativamente simples, só precisa de um pouco de engenharia reversa: entre na maquina de frontend e transforme o `wp-config.php` em um template. Para emular o final da configuração, você deve olhar no banco de dados criado pelo WordPress e preencher seu banco de dados de acordo. Existem playbooks para WordPress bem completos que oferecem esta opção.

Precisei modificar um pouco o role `mysql` para que ele trocasse o arquivo de configuração do MySQL por um que ouça em todas as interfaces de rede da máquina (originalmente, o MySQL só ouve na interface 127.0.0.1) e também uma correção para que o usuário WordPress tenha permissão para conectar de outras máquinas.

Poderíamos ter usado o Ansible para corrigir apenas esta linha no arquivo original mas minha recomendação é versionar seus arquivos de configuração da maneira que precisam ser no final e

variar parâmetros utilizando templates. Desta forma, dados como IPs, senhas, tokens e até conteúdo de banco de dados são facilmente identificados caso outra pessoa precise utilizar seu playbook.

Ao executar `vagrant up`, os dois blocos de código são executados em série com os respectivos provisionadores. O playbook mudou também: tem duas sessões marcadas pelo atributo `host`. Se fôssemos instalar em duas máquinas virtuais em provedores externos, teríamos que preencher um arquivo `hosts.ini` com dois grupos como no exemplo a seguir:

```
[web]
IP da VPS1
[db]
IP da VPS2
```

Executaríamos, portanto:

```
$ ansible-playbook -i hosts.ini blog.yml
```

Se a máquina estiver configurada corretamente para aceitar conexão utilizando chave SSH sem senha, o Ansible conectará em cada uma das máquinas e executará os playbooks.

5.3 INSTALANDO SEU WORDPRESS EM MÁQUINAS VIRTUAIS.

O exercício que proponho antes de prosseguir é escolher um provedor de infraestrutura e instalar este playbook em duas máquinas virtuais fora da sua máquina local. As informações até aqui são suficientes, você deve criar as duas máquinas manualmente e criar um `hosts.ini` com os endereços IPs corretos.

Para fazer isso na *DigitalOcean*, siga os passos do primeiro capítulo para criar uma conta, vá ao painel e crie uma chave SSH. A DigitalOcean tem um bom preço entre provedores de máquinas virtuais e não é difícil conseguir um cupom de descontos. Procure o

item *your settings* no seu profile e escolha o tab security (https://cloud.digitalocean.com/settings/security). Neste item procure *SSH Keys* e crie uma nova chave. Caso tenha dúvidas para criar uma chave SSH, consulte o mesmo capítulo.

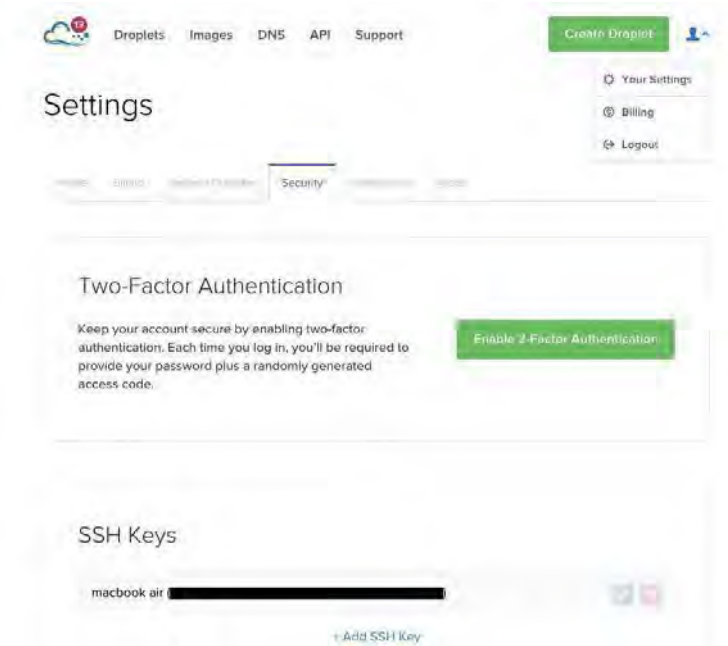


Figura 5.1: Painel de controle da DigitalOcean

Com a chave criada, vamos um fazer um playbook que cria duas máquinas com Ubuntu 14.04 e instala `web` e `db` em cada uma delas usando o Ansible. O Ansible tem módulos que interagem com provedores de serviço usando suas APIs.

A DigitalOcean lançou uma nova API em abril de 2015, portanto seu módulo ainda não havia sido atualizado no Ansible até a data de edição deste livro. Vou indicar as duas maneiras de acessá-la, mas o exemplo utilizará a V1 (versão 1) da API.

No painel da *DigitalOcean* vá até o item *API*:

Figura 5.2: API da DigitalOcean

Para a versão 1.0 da API, clique no link *View API v1*. Aparecerá uma tela para a criação de um par de *Client ID* e *Apy Key*. Clique em `Generate New Key` e copie os valores apresentados:

Figura 5.3: API versão 1 da DigitalOcean

Estes dois valores serão importantes para o provisionamento. Exporte os dois em variáveis de ambiente:

```
$ export DO_CLIENT_ID=<seu client id>
$ export DO_API_KEY=<sua api key.
```

Vamos testar com `curl` :

```
$ curl "https://api.digitalocean.com/v1/sizes/
```

```
?client_id=$DO_CLIENT_ID&api_key=$DO_API_KEY" |
python -m "json.tool"
```

Deve retornar um `json` com todos os tamanhos de máquinas virtuais da API. O módulo da DigitalOcean do Ansible depende de IDs de região, *SSH key*, imagem e tamanho para provisionar. Eu fiz um pequeno script para encontrar os IDs facilmente usando as variáveis de ambiente que foram declaradas, chamado `do_api_v1.py`:

```
"""
dependencias:
    sudo pip install dopy pyopenssl ndg-httpsclient pyasn1
"""

import os
from dopy.manager import DoManager
import urllib3.contrib.pyopenssl
urllib3.contrib.pyopenssl.inject_into_urllib3()


cliend_id = os.getenv("DO_CLIENT_ID")
api_key=os.getenv("DO_API_KEY")

do = DoManager(cliend_id, api_key)

keys = do.all_ssh_keys()
print "Nome da chave ssh\tid"
for key in keys:
    print "%s\t%d" % (key["name"], key["id"])

print "Nome da imagem\tid"
imgs = do.all_images()
for img in imgs:
    if img["slug"] == "ubuntu-14-04-x64":
        print "%s\t%d" % (img["name"], img["id"])

print "Nome da regiao\tid"
regions = do.all_regions()
for region in regions:
    if region["slug"] == "nyc2":
        print "%s\t%d" % (region["slug"], region["id"])

print "Nome do tamanho\tid"
sizes = do.sizes()
for size in sizes:
```

```
    if size["slug"] == "512mb":
        print "%s\t%d" % (size["slug"], size["id"])
```

Para o Ansible provisionar na DigitalOcean, precisamos de um módulo Python chamado *dopy*, além de bibliotecas auxiliares para o script de busca na API:

```
$ sudo pip install dopy pyopenssl ndg-httpsclient pyasn1
```

Execute o script para conferir os dados que serão completados no playbook. A versão que coloquei no livro e no repositório está atualizada mas os valores podem mudar.

```
$ python do_api_v1.py
```

Com a API e a chave SSH configurados, vamos ver um exemplo para criar e destruir uma máquina virtual usando o Ansible. Criamos um arquivo chamado `do_hosts.ini` :

```
[local]
localhost
```

Vamos usar um módulo *core* do Ansible para criar uma máquina:

```
- hosts: all
  connection: local
  user: ubuntu

  tasks:
    - name: "Cria uma máquina na DigitalOcean"
      digital_ocean: state=present
        command=droplet
        name=testevm
        size_id=66
        region_id=4
        image_id=9
        ssh_key_ids= 44
        virtio=yes
      register: testvm

    - name: "Id da máquina"
      debug: msg="{{ testvm.droplet.id }}"
```

```
    - name: "IP da máquina"
      debug: msg="{{ testvm.droplet.ip_address }}"
```

Os valores `66`, `4`, `9` indicam respectivamente tamanho da máquina, região e sistema operacional. O valor de `ssh_key_ids` deve ser o que vem da sua chamada da API. Estes dados indicam que o Ansible deve criar uma máquina na região *nyc2*, tamanho de 512mb, sistema operacional Ubuntu 14.04 64 bits e incluir a minha chave SSH de id 44 diretiva `debug` vai imprimir um `json` com o valor do atributo registrado na variável `testvm`. Anote este valor para preencher o playbook de remoção de máquina.

Depois de criada a máquina, verifique no seu painel se ela foi realmente criada e se conecte para testar. Um dos parâmetros impressos deve ser atualizado no playbook `remove_maquina.yml` no atributo ´id` para remover a máquina:

```
- hosts: all
  connection: local
  user: ubuntu

  tasks:
    - name: "Remove uma máquina na DigitalOcean"
      digital_ocean: state=deleted
        id=<id da maquina criada>
        command=droplet
        name=testevm
```

Para executar este playbook:

```
$ ansible-playbook -v -i do_hosts.ini remove_maquina.yml
```

Sabemos criar e remover uma máquina programaticamente. Se juntarmos com os playbooks `db.yml` e `web.yml`, teremos o playbook para instalar nosso WordPress com duas máquinas a seguir:

```
- hosts: all
  connection: local
  user: ubuntu

  tasks:
```

```
- name: "Cria a maquina web"
  digital_ocean: state=present
    command=droplet
    name=wordpress-web
    size_id={{ do_size_id }}
    region_id={{ do_region_id }}
    image_id={{ do_image_id }}
    ssh_key_ids={{ do_ssh_key_ids }}
    virtio=yes
  register: web

- name: "Cria a maquina db"
  digital_ocean: state=present
    command=droplet
    name=wordpress-db
    size_id={{ do_size_id }}
    region_id={{ do_region_id }}
    image_id={{ do_image_id }}
    ssh_key_ids={{ do_ssh_key_ids }}
    virtio=yes
  register: db

- name: "Espera confirmação antes de seguir, verificando a
         porta 22"
  wait_for: port=22 host="{{ web.droplet.ip_address }}"
    search_regex=OpenSSH delay=30

- name: "Espera confirmação antes de seguir, verificando a
         porta 22"
  wait_for: port=22 host="{{ db.droplet.ip_address }}"
    search_regex=OpenSSH delay=30

- name: "Id da máquina web"
  debug: msg="{{ web.droplet.id }}"

- name: "IP da máquina web"
  debug: msg="{{ web.droplet.ip_address }}"

- name: "Id da máquina db"
  debug: msg="{{ db.droplet.id }}"

- name: "IP da máquina db"
  debug: msg="{{ db.droplet.ip_address }}"

- name: "Adiciona host ao grupo instancias web"
  add_host: hostname={{ web.droplet.ip_address }}
    groupname=instancias_web
```

```
    - name: "Adiciona host ao grupo instancias db"
      add_host: hostname={{ db.droplet.ip_address }}
        groupname=instancias_db

- hosts: instancias_web
  sudo: True
  user: root
  roles:
    - common
    - php
    - nginx
    - wordpress

- hosts: instancias_db
  sudo: True
  user: vagrant
  vars_files:
    - vars/mysql.yml
  roles:
    - common
    - mysql
```

Para separar os dados gerados pela API, criei um diretório chamado `group_vars` e um arquivo chamado `all` dentro dele. Este diretório é lido pelo Ansible automaticamente e cada arquivo contém variáveis aplicadas a grupos específicos. O arquivo `all` contém variáveis aplicadas a todos os grupos de hosts.

```
do_size_id: 66
do_region_id: 4
do_image_id: 9
do_ssh_key_ids: <id da sua chave ssh>
```

A criação de máquinas virtuais poderia ter sido separada em um role mas preferi deixá-la assim para demonstrar a sequência de operações. Não é uma operação instantânea e a API pode demorar a responder. Para garantir a criação de máquinas íntegras, coloquei uma checagem extra da porta 22 TCP.

Para criar as duas máquinas e instalar os roles corretamente, execute:

```
$ export ANSIBLE_HOST_KEY_CHECKING=False
$ ansible-playbook -v -i do_hosts.ini wordpress_do.yml
```

O processo de configuração do WordPress continua o mesmo: pela interface web configure o IP da máquina e faça um post. Em alguns passos, a API pode responder com *timeouts* ou erros de SSH.

Como a remoção das máquinas envolvem seus IDs, eu recomendo fazê-lo pelo painel da DigitalOcean e apagar as entradas dos IPs criados do arquivo `~/.ssh/known_hosts`.

5.4 DIGITALOCEAN API V2

Tanto a API da DigitalOcean quanto o módulo do Ansible são novos e até a data da edição deste livro não haviam sido lançados oficialmente. Vamos ver como criar um token da nova API, e posteriormente a documentação do Ansible deve ser atualizada para permitir o uso da nova versão. Enquanto isso não acontece, recomendo que use a v1 da API como explicado anteriormente.

Na tela principal, vamos criar um *token*. A diferença é que o `CLIENT_ID` não será mais necessário.

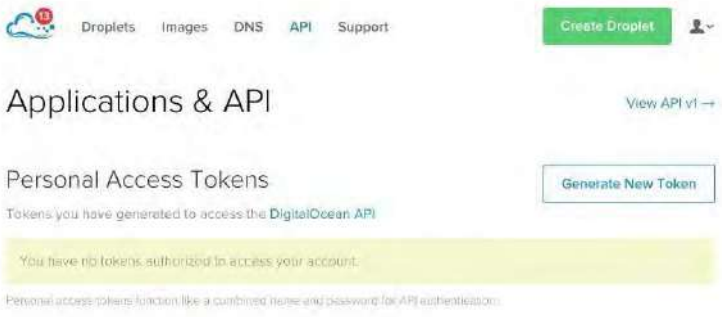


Figura 5.4: API da DigitalOcean

Clique em `Generate New Token` ao lado de *Personal Access Tokens*. A próxima tela terá um espaço para digitar o nome deste token e selecionar o escopo. Mantenha *Read* e *Write* marcados pois

precisaremos de permissões de escrita e leitura.

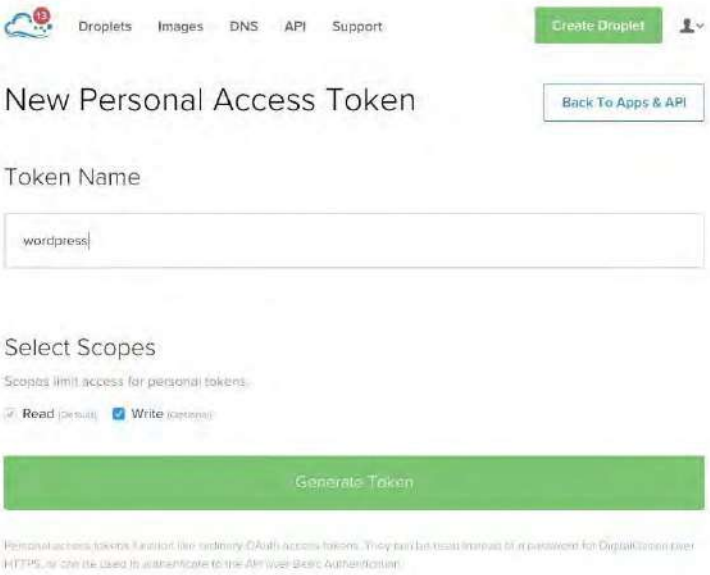


Figura 5.5: Criação do TOKEN da API da DigitalOcean v2

Quando gerar o token, será levado para outra tela com o resultado. Guarde seu token em um local seguro. Ele não aparecerá mais nesta tela por questões de segurança. O acesso que ele provê é o mesmo de uma senha que pode criar, remover e coletar detalhes de máquinas. Certifique-se de que ele não será inserido acidentalmente em nenhum repositório de código público ou privado.

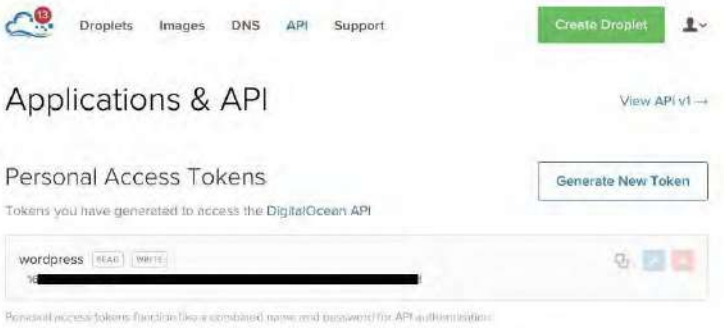


Figura 5.6: Token criado

Para testar, exporte o valor do seu token na variável `token` e use o `curl`:

```
$ export TOKEN=<seu token da api>
$ curl -X GET "https://api.digitalocean.com/v2/droplets" -H
"Authorization: Bearer $TOKEN"
```

Se você já tem alguma máquina na DigitalOcean, ela deve aparecer na resposta, caso contrário a resposta deve ser vazia. Se tiver dúvida, aumente o nível de verbosidade do `curl` adicionando o parâmetro `-vvv` antes do parâmetro `-X`.

Com a evolução deste módulo no Ansible, algumas mudanças devem acontecer na documentação. A mais importante será que em vez de utilizarmos *IDs* vamos utilizar o que é chamado de *SLUG*, um nome compacto para o recurso, por exemplo 512mb no lugar de 66.

5.5 CONCLUSÃO

Neste exemplo, as máquinas eram independentes. E se precisássemos instalar um software que funcione realmente em cluster, e que dependa de outros elementos deste cluster? No próximo capítulo vamos montar um playbook para instalar e

executar o Cassandra localmente e na AWS e esse exercício será útil.

Capítulo 6

CASSANDRA E EC2

Vamos criar um playbook para o banco de dados não relacional Cassandra. O Cassandra usa JAVA, tem um protocolo de comunicação entre o cluster por TCP/IP e precisa conhecer seus vizinhos. A arquitetura do Cassandra é parecida com outros sistemas distribuídos. A discussão e construção deste playbook e a execução local e na AWS vai nos ajudar a pensar melhor em como organizar nossas aplicações.

Neste playbook, vamos utilizar roles prontos para a instalação do JAVA e o repositório da *Datastax* para instalar o Cassandra. Será um cluster local de três máquinas virtuais, o que é realmente limitado para o Cassandra. Para experimentá-lo, você deve procurar máquinas com pelo menos 4GB de memória e duas *vcpus*.

O Cassandra é um banco de dados distribuído que, entre outras técnicas de sistemas distribuídos, utiliza um protocolo chamado *Gossip* para detecção de falhas e atualização do estado do cluster. Este protocolo é *peer-to-peer* (P2P), o que significa que cada nó deve conhecer todos ou pelo menos uma parte dos nós com os quais vai se comunicar.

Existe comunicação e troca de informações entre os nós mesmo quando não temos consultas ou escritas no banco de dados. Uma das condições com que o Gossip ajuda é no *Hinted Hand Off*, a guarda do estado de uma escrita por um nó e que pode ser entregue ao nó de destino assim que ele voltar ou ser remanejado para outro.

Vamos estruturar um role para a instalação do Cassandra que é flexível o suficiente para conhecer todos os nós que está subindo e prover a configuração correta para o cluster de acordo com a infraestrutura escolhida. Este dado é importante pois os nós do cluster devem ter as informações de seu ambiente disponível.

O Cassandra tem um componente com esta responsabilidade, chamado *Snitch*. Ele contém a lógica para localizar e classificar os nós para que a consistência e particionamento indicados em níveis mais altos do sistema seja respeitada. Vamos tornar a opção de Snitch flexível.

6.1 VAGRANT E CASSANDRA

Na configuração do Cassandra, precisamos enviar o IP em que o nó deve ouvir localmente e os IPs dos vizinhos, que chamaremos de *seeds*. Vamos usar o resto da configuração sem mudanças, mas se você for executar o Cassandra na AWS ou outro provedor de infraestrutura, teremos que mudar o Snitch (veja mais sobre os Snitchs disponíveis na documentação (http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html)).

O Snitch é um componente do Cassandra que descobre organiza a infraestrutura em que os nós são instalados, para saber em que *rack* e *datacenter* as réplicas dos dados serão armazenadas. Para infraestruturas locais e datacenters tradicionais, podemos usar o *DynamicSnitching* ou o *SimpleSnitch*. Para a AWS usaremos o *EC2Snitch*.

É o Snitch que descobre ou tem as definições de datacenter, rack e redes que serão aproveitadas na definição da estratégia de replicação do keyspace, além do protocolo *Gossip*. Em arquiteturas em que a máquina não possui o IP válido em uma de suas interfaces

(como na AWS), a comunicação entre datacenters é feita por outros canais e o Snitch é responsável por estas informações. Vamos mudar esta configuração quando instalarmos o mesmo cluster na AWS.

Para instalar o Cassandra, vamos precisar de um role para instalar Java que não seja o OpenJDK. Eu escolhi este role (https://github.com/vrischmann/ansible-role-java) por ser completo e simples de utilizar. O role de instalação de Cassandra é meu e além disso instalo suporte a Python nas máquinas, pois é o que eu costumo usar. A estrutura de diretórios será parecida com a última versão do playbook do WordPress.

O `Vagrantfile` deste playbook usa Ruby para facilitar nosso trabalho de deploy:

```
# -*- mode: ruby -*-
# vi: set ft=ruby :

VAGRANTFILE_API_VERSION = "2"

# Prepara os dados para todos os nodes do cluster
nodes = ['192.168.33.100', '192.168.33.101', '192.168.33.102']
cluster_name = "Vagrant Cluster"

servers = []
nodes.each_with_index do |node, idx|
  servers << {
    'hostname' => 'node' + idx.to_s,
    'ip' => node,
    'seeds' => nodes.join(","),
    'cluster_name' => cluster_name,
  }
end

# Para cada item da lista de servidores (servers) uma vm
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
    servers.each do |server|
        config.vm.define server['hostname'] do |cfg|
            cfg.vm.box = "ubuntu/trusty64"
            cfg.vm.host_name = server['name']
            cfg.vm.network :private_network, ip: server['ip']
            cfg.vm.provision "ansible" do |ansible|
                ansible.extra_vars = {
```

```
                    cluster_name: server["cluster_name"],
                    seeds: server["seeds"],
                    listen_address: server['ip'],
                    rpc_address: server['ip']
                }
                ansible.verbose = 'vvvv'
                ansible.playbook = "cassandra.yml"
            end
            cfg.vm.provider "virtualbox" do |v|
                v.customize ["modifyvm", :id, "--memory", "2048"]
            end
        end
    end
end
```

A lista `nodes` contém 3 endereços IP, com base nos quais vamos instruir o Vagrant a criar e provisionar estes nós. Para cada máquina, vamos passar as variáveis que o role de instalação do Cassandra vai utilizar para configurar cada nó.

O atributo `extra_vars` do Ansible aceita um dicionário (Hash em Ruby) com valores que serão repassados como variáveis do Ansible. O bloco dentro de `servers.each` repete o processo de criação e provisionamento de máquinas virtuais para cada item de `servers`.

A preparação inicial da lista `servers` usa nodes para criar nomes para cada um dos elementos do cluster (node0, node1 e node2), dá um endereço IP para cada nó e cria o parâmetro `seeds` com todos os IPs, além do nome do cluster.

Esta é a configuração mínima para o Cassandra, entre as centenas de itens que podem ser modificados. Com isso, criamos um cluster de 3 máquinas. Não instalei o *opscenter* nem outras ferramentas necessárias para executar o Cassandra em ambiente de produção, apenas o banco de dados.

Esta configuração fica pesada em um computador normal, pois as máquinas precisam de pelo menos 2GB de memória para evitar

que o kernel do Linux rode o OOM Killer (*out of memory killer*). Poderíamos habilitar o *swap* mas sairia do escopo do provisionamento. É um exercício que eu sugiro: melhorar a configuração destas máquinas virtuais e criar as tasks no playbook.

O teste para cada nó é simples: `vagrant ssh node[0..2]` e execute o comando `nodetool status` para ver o estado do nó. Este playbook completo pode ser encontrado no repositório do livro.

Para instalar este cluster fora do Vagrant vamos precisar das variáveis para cada nó. Veja como fica o inventário `hosts.ini` para este caso, assumindo que cada um dos IPs é uma maquina pronta e com o SSH configurado para autenticar sem senha por chave pública:

```
[cassandra]
192.168.33.100 cluster_name="Super Cluster"
  seeds="192.168.33.100,192.168.33.101,192.168.33.102"
  listen_address="192.168.33.100" rpc_address="192.168.33.100"
192.168.33.101 cluster_name="Super Cluster"
  seeds="192.168.33.100,192.168.33.101,192.168.33.102"
    listen_address="192.168.33.101" rpc_address="192.168.33.101"
192.168.33.102 cluster_name="Super Cluster"
  seeds="192.168.33.100,192.168.33.101,192.168.33.102"
  listen_address="192.168.33.102" rpc_address="192.168.33.102"
```

Para executar o Ansible nas três máquinas:

```
ansible-playbook -i hosts.ini cassandra.yml
```

Você poderia gerar dinamicamente este arquivo vindo de um sistema de gerenciamento de máquinas ou de configuração. O Ansible chama isso de *dynamic inventory* e é um dos principais recursos do *Ansible Tower*, a solução centralizada de gerenciamento que o Ansible oferece.

Existem plugins para gerenciar o inventário dinâmico sem usar Ansible Tower. Escolhi a AWS para testar este playbook. Vamos instalar 3 máquinas para o Cassandra, mas antes vamos examinar

somente o provisionamento.

6.2 PROVISIONAMENTO NA AWS

O Ansible tem o código e módulos necessários para utilizar a API da AWS. Mas como provisionador ele deve manter um conjunto de informações sobre o inventário de máquinas que estamos utilizando. A AWS faz isso do lado da API e precisamos de meios de recuperar esta informação em um formato que o Ansible compreenda.

Precisamos do plugin de inventário dinâmico para EC2 instalado. Este plugin nada mais é que um programa que facilita a comunicação com a API do EC2 e substitui o `hosts.ini`. Para que ele funcione, você vai precisar de uma conta na AWS, credenciais (`AWS_ACCESS_KEY_ID` e `AWS_SECRET_ACCESS_KEY`) e a chave pública/privada de SSH configurada. As credenciais são encontradas no menu IAM do painel da AWS. Vamos seguir as instruções do Ansible:

```
$ sudo pip install boto # or use your favorite package manager
$ curl https://raw.githubusercontent.com/ansible/ansible/devel/
plugins/inventory/ec2.py > ec2.py
$ chmod +x ec2.py
$ curl https://raw.githubusercontent.com/ansible/ansible/devel/
plugins/inventory/ec2.ini > ec2.ini
$ export AWS_ACCESS_KEY_ID=seu access key id
$ export AWS_SECRET_ACCESS_KEY=seu secret access key
$ ./ec2.py --help
```

Anote o nome da sua chave na AWS. Se não tiver chave, no console, menu EC2, vá ao item *Key Pairs* e crie uma. Um arquivo com extensão `.pem` será salvo em sua máquina. A minha tem o nome de `aws_devel`, portanto o arquivo se chama `aws_devel.pem`.

Para usar esta chave com o SSH, copie para seu diretório `~/.ssh/` e mude as permissões para 0600 com `chmod 0600`

`~/.ssh/aws_devel.pem` . Quando a máquina estiver instalada, teste com `ssh -i ~/,ssh/aws_devel.pem ubuntu@ip_publico` , trocando `aws_devel.pem` pelo nome do arquivo com a sua chave.

Dentro do diretório `group_vars` criamos um arquivo chamado `all` com os dados de chave SSH, região, nome da imagem, VPC (*Virtual Private Cluster*), *subnet* e tipo de instância. O diretório `group_vars` contém arquivos com o nome dos grupos e `all` para variáveis que se aplicam para todos os grupos.

```
key_name: aws_devel
aws_region: us-east-1
ami_id: ami-9
instance_type: t2.micro
vpc_id: vpc-XXXXXXXX
subnet_id: subnet-XXXXXXX
```

Esta é a descrição para máquinas do EC2 na região us-east-1, sistema operacional Ubuntu 14.04 t2.micro (a t1.micro é apenas PV e a imagem que queremos usar é HVM. São tipos de virtualização que impactam na performance e tamanho da máquina). Vamos precisar de uma VPC criada e uma subnet atrelada a esta VPC. Existe um *wizard* para isso e provavelmente você já está usando VPC e subnets.

Para se familiarizar com o Ansible como provisionador, criei dois playbooks que executam a tarefa que já fizemos anteriormente: instalar uma máquina com *nginx*. Vamos fazer um desvio aqui para executar este playbook e conferir se tudo está configurado corretamente. Execute o comando:

```
$ ansible-playbook -vvvv -i aws_hosts.ini cria_maquina.yml
--private-key ~/.ssh/aws_devel.pem
```

O parâmetro `--private-key` deve apontar para seu arquivo com a chave pública, o parâmetro `-vvvv` serve para nos mostrar todos os passos que estão sendo executados. O arquivo `aws_hosts.ini` aponta para localhost, pois a maioria das tarefas serão executadas **a**

partir da máquina local.

A construção deste playbook é interessante pois tem duas sessões: o provisionamento da máquina na AWS e o provisionamento na máquina já criada. Introduzimos também a task `wait_for` para esperar a criação da máquina virtual.

Como a criação de máquinas virtuais é remota, o tempo pode variar mais do que a criação local com o Vagrant. Esta primeira sessão substitui algumas tarefas do Vagrant ao coletar dados da máquina criada e repassar para a segunda sessão, que contém a task que instala o *nginx*.

Vamos examinar cada task do provisionamento:

```
# Cria uma máquina
- hosts: localhost
  connection: local
  gather_facts: False
  tasks:
  - name: "Cria security group"
    ec2_group:
      name: cassandra_group
      description: "Cassandra Security group"
      vpc_id: "{{vpc_id}}"
      region: "{{aws_region}}"
      rules:
        - proto: tcp
          type: ssh
          from_port: 22
          to_port: 22
          cidr_ip: 0.0.0.0/0
        - proto: tcp
          type: http
          from_port: 80
          to_port: 80
          cidr_ip: 0.0.0.0/0
      rules_egress:
        - proto: all
          type: all
          cidr_ip: 0.0.0.0/0
    register: ec2_firewall
```

O início deste playbook indica que estas tarefas serão executadas

localmente. Com a ajuda da biblioteca *boto* em Python, o Ansible pode usar a API da AWS para criar máquinas e outros serviços. O passo `gather_facts: False` instrui o Ansible a não coletar informações do ambiente local.

A primeira task cria um *Security Group* chamado `cassandra_group` no *VPC* e região indicadas.

Para este grupo são criadas algumas regras de firewall para liberar o acesso a SSH e HTTP. *Egress* e *Ingress* são palavras indicadas para descrever o sentido dos pacotes de redes. *Egress* significa pacotes de rede que saem da máquina. Por último, a diretiva `register` coloca o resultado desta tarefa em uma variável global do playbook que pode ser acessada por outras tasks.

```
- name: "Cria uma instancia no EC2"
    local_action: ec2 key_name="{{key_name}}"
      vpc_subnet_id="{{subnet_id}}"
      region="{{aws_region}}"
      group_id="{{ec2_firewall.group_id}}"
      instance_type="{{instance_type}}"
      image="{{ami_id}}"
      wait=yes
      assign_public_ip=yes
    register: ec2
```

Nesta tarefa, criamos a máquina virtual. Criei uma máquina e além dos dados de chave SSH (`key_name`), `subnet_id` , região da AWS (`region`), imagem e tipo de instância, passei o `group_id` que foi gravado dentro da variável `ec2_firewall` vinda da tarefa anterior. Esta instância terá um IP válido (`assign_public_ip=yes`).

A task está configurada para esperar a conclusão da chamada da API com `wait=yes` . Se estivéssemos em um VPC que usa uma máquina de gateway, poderíamos criar as máquinas sem IP válido e utilizar apenas a rede interna do VPC. Registramos o resultado da task na variável `ec2` .

```
- name: "Adiciona host ao grupo instancias"
```

```
  add_host: hostname={{ item.public_ip }} groupname=instancias
  with_items: ec2.instances

- name: "Espera confirmação antes de seguir, verificando
         a porta 22"
  wait_for: port=22 host="{{ item.public_ip }}"
    search_regex=OpenSSH delay=10
  with_items: ec2.instances
```

A variável ec2 serve para a próxima task `Adiciona host ao grupo instancias` criar uma lista de máquinas que posteriormente usaremos como hosts para a segunda parte do playbook.

É importante entender que este inventário não existe em um arquivo pois ainda estamos criando as máquinas. O artifício para tanto é usar um auxiliar externo, como o `ec2.py` e também utilizar os dados que são retornados ao final de cada task.

A última task da primeira parte é uma precaução para esperar a criação e boot da máquina virtual. Ela apenas espera que a porta 22 (SSH) esteja aberta e que devolva uma string com "OpenSSH" após ter uma conexão. Pode testar diretamente do seu console com um `telnet ip_da_maquina 22` e você verá que antes do trfego criptografado o servidor SSH manda um breve cabeçalho com esta string.

```
# Usando a máquina que criamos, vamos instalar o NGINX.
- hosts: instancias
  sudo: True
  user: ubuntu
  gather_facts: True
  tasks:
    - name: "Atualiza pacotes e instala nginx"
      apt: name=nginx state=latest update_cache=yes
      install_recommends=yes
```

Na segunda parte tudo ficou simples: usamos a variável `instancias` que criamos a partir do IP público vindo do resultado da execução da task `Cria uma instancia no EC2` e utilizamos como um playbook normal. Este é o inventário dinâmico dentro da

mesma rodada do playbook.

Se você executar o playbook novamente, ele não agirá sobre instâncias criadas em outras execuções. O papel do `ec2.py` ficará mais claro ao analisarmos o playbook de remoção de máquina. Antes de executá-lo, faça:

```
$ ./ec2.py --list
```

O retorno do comando será um `JSON` com todas as máquinas criadas e seus status. Explore as outras opções deste comando trocando `--list` por `--help`. A maneira de fazer um playbook "lembrar" do estado de execuções anteriores para a AWS é usar o `ec2.py` como seu `hosts.ini` (*inventory file*). Vamos utilizá-lo para remover a máquina que criamos. Antes de apagá-la, teste o SSH e o webserver. Para se conectar à máquina use:

```
$ ssh -i ~/.ssh/aws_devel.pem ubuntu@ip_publico_da_sua_máquina
```

Execute novamente o programa `ec2.py` (`./ec2.py`) e examine sua saida. O nome de grupo que colocamos anteriormente aparece com o prefixo `security_group`. Esta sessão é interpretada como uma sessão de inventário estático. Portanto, conseguimos executar tasks nas máquinas que estão neste grupo. É o que a primeira parte do playbook faz. Vamos remover a máquina antes e posteriormente examinar o playbook.

```
$ ansible-playbook -i ./ec2.py remove_maquina.yml
```

```
- hosts: security_group_cassandra_group
  connection: local
  gather_facts: False
  tasks:
    - name: Remove a instância
      local_action:
        module: ec2
        state: 'absent'
        region: '{{aws_region}}'
        instance_ids: '{{ec2_id}}'

- hosts: localhost
```

```
connection: local
gather_facts: False
tasks:
  - name: Remove o security group cassandra_group
    local_action:
      description: "Cassandra group"
      module: ec2_group
      name: security_group_cassandra_group
      region: "{{aws_region}}"
      state: 'absent'
```

Neste playbook, inicialmente removemos a máquina e posteriormente o `security group`. Poderíamos não tê-lo removido mas optei por deixar este ciclo completo. Note que as variáveis que estão no arquivo `groups_var/all` são aplicadas às tarefas também, pois pertencem a todos os grupos e todos playbooks deste diretório.

Em ambas as tarefas usamos o estado (*state*) `absent` para indicar que aquela descrição indica um item que deve estar ausente (removido) ao final da tarefa. Esta semântica se repete para todas as tarefas do Ansible: instalação de pacote, movimentação de arquivos, templates e provisionamento.

6.3 CASSANDRA E AWS

Agora estamos prontos para retomar o provisionamento do Cassandra. Examine o playbook para ver como integrei o que tínhamos para uso com o Vagrant com o que aprendemos sobre EC2. Criei um playbook separado para facilitar a comparação. Este playbook vai parecer grande pois terá as funções de provisionamento e configuração de ambiente, mas com o que vimos não parecerá complexo.

A descrição de nossa tarefa é combinar os playbooks anteriores, criar uma configuração de firewall que permita que os nós de Cassandra se comuniquem, criar máquinas um pouco maiores para que o sistema consiga ser executado e fazer com que as máquinas se

comuniquem pela rede interna. Temos que extrair e organizar as informações das instâncias criadas para satisfazer os requerimentos das variáveis dos roles que vamos usar.

Este playbook está no mesmo repositório com o nome de `cassandra_aws.yml`. Não esqueça de desligar suas máquinas quando terminar de usá-las pois o custo pode ser alto.

Usei instâncias menores do que as recomendadas para a operação segura de um cluster Cassandra pois meu objetivo é mostrar o provisionamento. Vamos examinar o `hosts.ini` do provisionamento local:

```
[cassandra]
192.168.33.100 cluster_name="Super Cluster"
  seeds="192.168.33.100,192.168.33.101,192.168.33.102"
  listen_address="192.168.33.100" rpc_address="192.168.33.100"
192.168.33.101 cluster_name="Super Cluster"
  seeds="192.168.33.100,192.168.33.101,192.168.33.102"
  listen_address="192.168.33.101" rpc_address="192.168.33.101"
192.168.33.102 cluster_name="Super Cluster"
  seeds="192.168.33.100,192.168.33.101,192.168.33.102"
  listen_address="192.168.33.102" rpc_address="192.168.33.102"
```

Temos que criar as variáveis `cluster_name`, `seeds`, `listen_address` e `rpc_address`, cada máquina virtual utilizando os dados de criação de cada instância. Criamos uma task para acumular seeds e modifiquei o código do role `cassandra`, no template `cassandra.yaml.j2` usei uma diretiva do Jinja2 para criar um valor default baseado em uma variável do inventário do Ansible. Comparando antes e depois:

```
Antes: listen_address: {{ listen_address }}
Depois: listen_address: {{ listen_address |
default(ansible_default_ipv4.address) }}
```

Para instâncias com IP interno, como a AWS em um VPC na mesma região, esta configuração junto com o *EC2Snitch* são as corretas. Pode ser diferente em outros provedores que forneçam apenas uma interface com IP válido ou entre regiões da AWS com o

EC2MultiRegionSnitch que demanda mudanças de regras na firewall e no VPC. Aproveitei a mesma técnica para a definição de Snitch, com um valor default para o `SimpleSnitch` .

```
endpoint_snitch: {{ snitch | default(SimpleSnitch) }}
```

Para este cluster vamos criar três instâncias, portanto criei a task de criação com o parâmetro extra `*count=3*` . O Ansible consegue provisionar as máquinas virtuais em paralelo. Vamos mudar o parâmetro `snitch` para `EC2Snitch` . Snitch no Cassandra é o componente que ajuda a gerenciar a infraestrutura. Nosso playbook tem uma jogada ninja para montar o *seeds*: criamos um grupo de hosts com os IPs privados e posteriormente o utilizamos para criar o parâmetro seeds.

```
- name: "Cria instancias no EC2 para o cluster"
    local_action: ec2 key_name="{{key_name}}"
      count="{{ vm_count }}"
      vpc_subnet_id="{{subnet_id}}"
      region="{{aws_region}}"
      group_id="{{ec2_firewall.group_id}}"
      instance_type="{{instance_type}}"
      image="{{ami_id}}"
      wait=yes
      assign_public_ip=yes
    register: ec2

  - name: "Adiciona host ao grupo instancias"
    add_host: hostname={{ item.public_ip }} groupname=instancias
    with_items: ec2.instances

  - name: "Cria a variável seeds com os IPs privados das
            instâncias"
    add_host: hostname={{ item.private_ip}} groupname=private_ips
    with_items: ec2.instances

vars:
    - java_versions: oracle-java7-installer
    - cluster_name: "Cassandra Cluster AWS"
    - snitch: EC2Snitch
    - seeds: "{{ groups['private_ips'] | join(',') }}"
```

Utilizamos o template engine Jinja2 embutido no Ansible para

acessar o grupo chamado `private_ips` preenchido na task `Cria a variável seeds com os IPs privados das instâncias`. Estes IPs, por suas vezes, foram extraídos das variáveis `ec2` geradas por cada provisionamento.

A diretiva `with_items` itera sobre a lista contida em `ec2` e para cada item extraímos o IP privado. Fizemos o mesmo para o IP válido para gerar o grupo chamado `instancias`. Após executar o playbook, navegue no console da AWS para ver suas instâncias, security groups e regras de firewall.

Existem mais dados e detalhes incluindo o dimensionamento das máquinas para executar o Cassandra. Eu recomendo o guia da Datastax (http://www.datastax.com/documentation/cassandra/2.1/cassandra/install/installAMISecurityGroup.html). Dependendo da sua configuração de SSH, pode aparecer uma mensagem reclamando sobre "known_hosts" parecida com esta:

```
The authenticity of host '111.111.111.111 (111.111.111.111)'
can't be established. RSA key fingerprint is
  :  :  :ff:ff:ff:ff:ff:  :  :  :  :  :  :ff:ff.
Are you sure you want to continue connecting (yes/no)?
```

Isso pode ser resolvido seguindo as instruções em (http://docs.ansible.com/intro_getting_started.html#host-key-checking) ou com `ssh-keygen`.

No final, o arquivo `group_vars/all` ganhou novos parâmetros para indicar o bloco de IPs e o número de VMs criadas (`count_vm`).

```
key_name: aws_devel
aws_region: us-east-1
ami_id: ami-9
instance_type: t2.micro
vpc_id: vpc-5
subnet_id: subnet-d9fffff
vm_count: 3
cidr_ip: 10.0.0.0/16
```

6.4 CONCLUSÃO

Neste capítulo, vimos como utilizar o Ansible como provisionador, como usar seu driver de AWS e EC2 e como trabalhar com um inventário dinâmico.

A dificuldade envolvida é entender os conceitos de provedores externos e das ferramentas que você usa.

O exercício que sugiro é portar o que fizemos para outro provedor como DigitalOcean e também executar os playbooks anteriores na AWS. Não esqueça de desligar suas máquinas se você gosta do seu cartão de crédito.

Capítulo 7

MÉTRICAS E MONITORAÇÃO

A diferença entre coletar métricas e monitorar uma aplicação é sutil: o mecanismo utilizado pode ser semelhante, mas o tempo e as ações tomadas são diferentes.

Na coleta de métricas, você cria visibilidade de como sua aplicação realmente se comporta em tempo real — não são testes sintéticos com um roteiro gerado previamente.

Você pode notar mudanças de tempo de resposta e degradação de partes modificadas de uma aplicação após cada deploy. As ações são tomadas em tempo de desenvolvimento e em produção para prevenção e diagnóstico.

A monitoração tradicional usa ferramentas que executam testes em intervalos predefinidos de tempo e toma conclusões como "serviço lento" ou "serviço indisponível". As ações são tomadas apenas em tempo de produção, sendo paliativas ou corretivas para manter o sistema funcionando.

Métricas são o caminho para entender como sua aplicação se comporta entre deploys e mudanças de arquiteturas. Existem maneiras de coletar métricas com instrumentação de código e com a coleta de dados do ambiente da aplicação.

Entre estes dois conceitos, existe o de *profiling*, que é o exame de

métrica de aplicações divididas por camadas para avaliar caminhos de código e problemas de capacidade.

As ferramentas para *profile* geralmente se conectam ao interpretador ou a camadas do sistema operacional para capturar a interação do código e gerar métricas.

O melhor dos mundos é entender como combinar dados de monitoração, métricas e profiling, além de eventos do dia a dia como mudanças e problemas de infraestrutura.

Este é o estado da arte de uma organização e um trabalho complexo. Podemos estudar bons exemplos de cada tipo de ferramentas e implementar o básico para nossas necessidades. Muitas ferramentas legais podem ser utilizadas com baixo ou nenhum custo.

Procure também material de conferências como Velocity (http://velocityconf.com/) e vídeos da Monitorama em (https://vimeo.com/monitorama/videos). Estes vídeos apresentam casos de implementação, construção de dashboards e desenvolvimento de métricas para detectar a degradação de serviços.

7.1 MONITORAÇÃO

A monitoração funcional nasceu do *ping*. A maioria dos sistemas disponíveis hoje em dia nasceu de antecessores ou combinações do Nagios (http://www.nagios.org/). Cada etapa de teste se chama *probe* e começa por executar ping, testar portas TCP/IP abertas, a resposta de sites e a existência de strings em logs.

Este estilo de monitoração tradicional se baseia em servidores centrais que executam os probes em intervalos predefinidos de tempo e toma decisões como enviar e-mail e guardar os valores

testados para análise posterior em um banco de dados RRD (*Round Robin Database*). Quando apresentam estas interfaces, estes sistemas têm descendência direta do MRTG (http://oss.oetiker.ch/mrtg/), por exemplo Cacti (http://www.cacti.net/).

Apesar de terem um valor imenso e serem utilizados até hoje, não vou me concentrar nestes sistemas para criar playbooks ou instalações completas. Vou indicar um playbook e deixar como exercício criar uma instalação de *check-mk* se o leitor tiver curiosidade nestes tipos de monitoração, *Open Monitoring and Check-MK* (https://github.com/sfromm/ansible-omdistro).

Em vez disso vamos olhar para serviços de monitoração no estilo *SaaS* (*Software as a Service*). O mais famoso deles é o Pingdom (https://pingdom.com) que oferece free trial.

O que eu gosto de usar por ser completo e iniciar com uma camada grátis além de boa progressão de preço é o Uptime Robot (https://uptimerobot.com/). É grátis para checagens de 5 em 5 minutos de até 50 itens. Eles cobram para intervalos menores, envio de SMS e retenção de histórico. Em ambos os planos você pode usar uma API para inclusão e exclusão de sites.

Este é meu dashboard configurado para poucos sites:

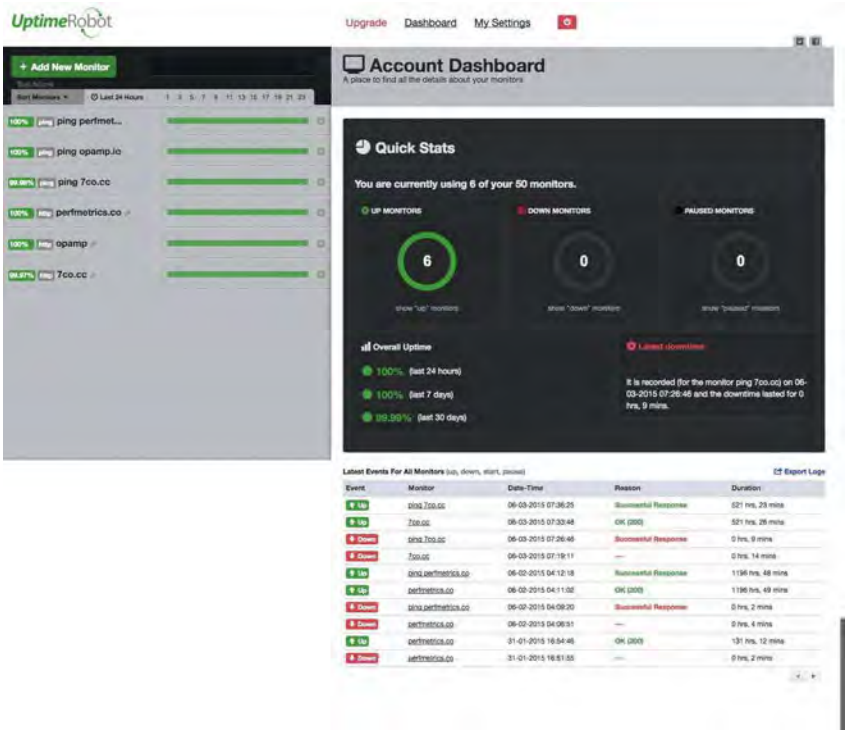


Figura 7.1: Dashboard exemplo do UptimeRobot

Este dashboard é parecido com o de outras ferramentas e contém 3 sessões principais: seus probes/Monitors no lado superior esquerdo, Quick Stats (um resumo visual rápido no lado superior direito) e no inferior direito uma lista dos últimos eventos (quedas, erros e máquinas reestabelecidas).

Vamos criar um probe para https://google.com. Caso você já tenha algum site, apenas troque o endereço pelo do seu site.

Entre com sua conta e clique o botão no alto, lado esquerdo com o título `Add New Monitor` . Deve aparecer a tela em seguida:

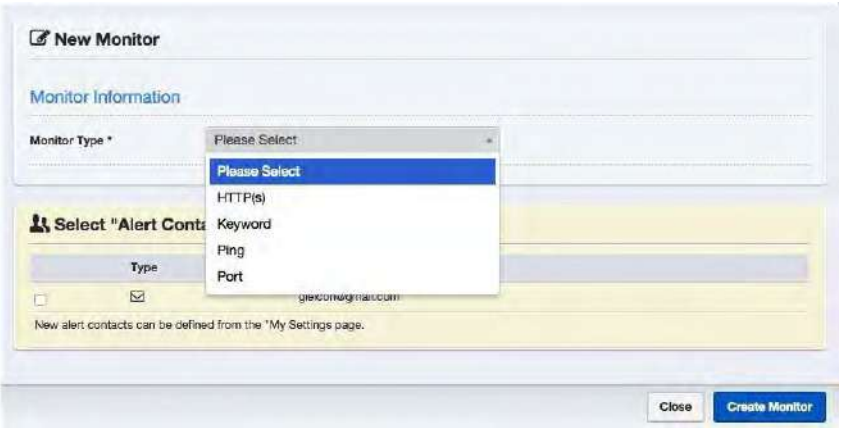


Figura 7.2: Add new monitor

Selecione *Monitor type* para HTTP e a tela deve mudar:

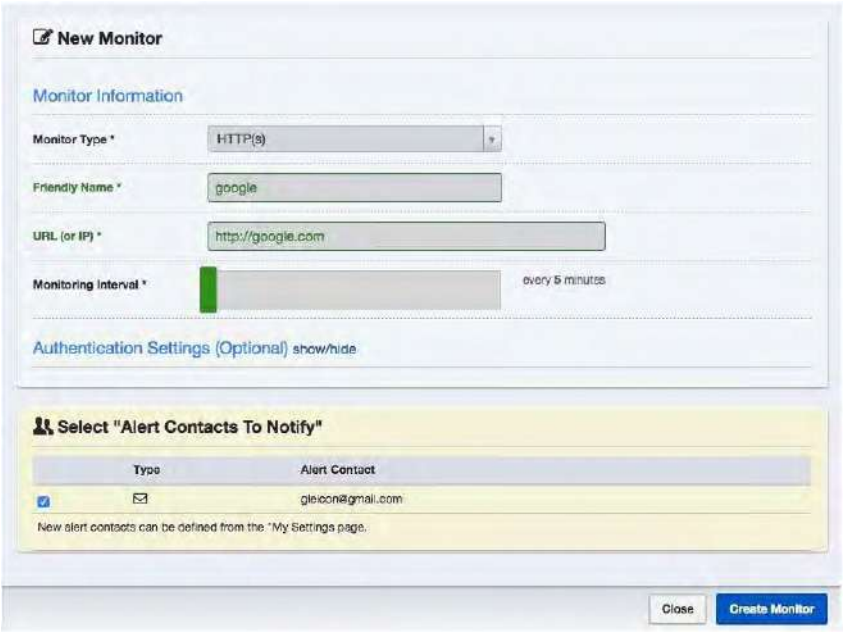


Figura 7.3: Seleciona o endereço

Preencha com o domínio do seu site ou com

"https://google.com" e clique o botão `Create Monitor`.

O Ansible oferece um pacote de módulos extra (https://github.com/ansible/ansible-modules-extras) que tem suporte ao UptimeRobot, entre outros. Podemos utilizar este módulo para parar e reiniciar a monitoração de um site criado pela interface web.

Para utilizar estes módulos, você deve ter a última versão do Ansible (a que estou usando agora é a 1.9, `HEAD` do repositório Git). Com a monitoração criada, vá em `Settings -> API Settings -> Create the main API key`, até receber a mensagem *The main API key is: x111111-1111111111111111*.

Com a chave de API criada, procure o ID de sua monitoração. É o número que fica após da URL quando você clica em cima do nome, por exemplo:

```
https://uptimerobot.com/dashboard#77
```

O ID do monitor é 77. Copie o arquivo `vars/uptimerobot_creds.yml.dist` para o arquivo `vars/uptimerobot_creds.yml` e altere os dados. Execute o Ansible para pausar a monitoração:

```
$ ansible-playbook -i hosts.ini uptimerobot_pause.yml
```

O código é simples:

```
- hosts: localhost
  connection: local
  vars_files:
    - vars/uptimerobot_creds.yml

  tasks:
  - name: "Pausa a monitoração do item "
    uptimerobot: monitorid="{{ monitor_id }}"
      apikey="{{ api_key }}"
      state=paused
```

Reinicie a monitoração com:

```
$ ansible-playbook -i hosts.ini uptimerobot_start.yml
```

Para este caso, o estado (*state*) muda para `started`. Estas tasks podem ser colocadas no início e no final do provisionamento para evitar falsos positivos ou alertas errados.

Para efeitos de monitoração tradicional de uptime, ping e porta aberta, vamos parar por aqui. A maioria das ferramentas web será parecida com o Uptime Robot portanto encontre a sua e procure um módulo ou maneira de automatizar a criação e o controle do seu monitor.

7.2 COLLECTD E LOGSTASH

No repositório deste capítulo tem um playbook chamado `logstash`. Este playbook é um exemplo de instalação do CollectD (http://collectd.org) e do Logstash (http://logstash.net/). O *collectd* é um framework de agentes e servidores para coleta de métricas e contadores.

Ele tem uma biblioteca de plugins que cobrem sistemas operacionais, aplicações e APIs. O *logstash* é um sistema de processamento e indexação de logs e métricas com um construtor de dashboards bem flexível. Ele usa o Elasticsearch (https://www.elastic.co/products/elasticsearch), um motor de busca e indexação poderoso baseado no *Apache Lucene*.

Estas funcionalidades são encontradas em produtos comerciais como o Splunk (http://www.splunk.com/), mas com a combinação de projetos de código aberto a flexibilidade de adicionar componentes e de limites de uso é maior.

A máquina virtual criada é autocontida, serve para examinar o *collectd* e o *logstash*. O IP dela é *192.168.33.100*. Após o comando

`vagrant up`, coloque o endereço http://192.168.33.100/ em seu browser e navegue pela interface web do *logstash*, construída em cima do Kibana (https://www.elastic.co/products/kibana).

A esta altura, você deve saber o que precisa fazer para instalar um playbook em uma máquina de produção. O *collectd* deve ser instalado em todas as máquinas que você quer monitorar, e a sua configuração em `/etc/collectd/collectd.conf` deve ser parecida com a seguinte:

```
Hostname "nome_da_sua_maquina"

FQDNLookup false
LoadPlugin cpu

LoadPlugin df
<Plugin df>
        Device "/dev/sda1"
        MountPoint "/"
        FSType "ext4"
        ReportReserved true
</Plugin>

LoadPlugin interface
<Plugin interface>
        Interface "eth0"
        Interface "eth1"
        IgnoreSelected false
</Plugin>

LoadPlugin network
<Plugin network>
  <Server "192.168.33.100" "25   "> # ip do servidor com logstash
  </Server>
</Plugin>

LoadPlugin memory

LoadPlugin syslog
<Plugin syslog>
        LogLevel info
</Plugin>

LoadPlugin swap
```

```
<Include "/etc/collectd/collectd.conf.d">
        Filter "*.conf"
</Include>
```

Você pode aproveitar o mesmo playbook para os clientes, só deve modificá-lo para colocar o IP do servidor com *logstash* no bloco indicado:

```
LoadPlugin network
<Plugin network>
  <Server "192.168.33.100" "25   "> # ip do servidor com logstash
  </Server>
</Plugin>
```

O plugin *network* envia os dados coletados para um servidor. No framework do *colelctd* são definidos transportes e armazenamentos de dados. Ele grava os dados coletados em arquivos `.rrd` locais e envia as métricas por este plugin. O *logstash* está configurado em `/etc/logstash/conf.d/logstash.conf` com um módulo que recebe o protocolo do *collectd*:

```
input {
  udp {
    port => 25
    buffer_size => 1452
    codec => collectd { }
    type => "collectd"
  }
}
output {
  elasticsearch {
    cluster => "dashboard"
    protocol => "http"
    host => "192.168.33.100:92  "
  }
}
```

São definidos dois blocos, um de entrada (*input*) e outro de saída (*output*). O *logstash* suporta mais blocos de entrada e saída simultaneamente. No bloco `output`, o parâmetro `host` aponta para o servidor *Elasticsearch*. No nosso caso, será a mesma máquina, mas você poderia ter um cluster de `elasticsearch` para conseguir alta

disponibilidade. Com esta máquina virtual, siga os passos para explorar o elasticsearch (http://www.elastic.co/guide/en/elasticsearch/reference/current/_exploring_your_cluster.html). Ele pode ser útil em outros projetos.

Você pode construir um sistema com a mesma funcionalidade utilizando SOLR (http://lucene.apache.org/solr/), InfluxDB (http://influxdb.com/), Grafana (http://grafana.org/), StatsD (https://github.com/etsy/statsd/) ou Graphite (http://graphite.wikidot.com/). Pode experimentar também o GrayLog (https://www.graylog.org/). O objetivo desta lista é dizer que o importante é a ideia principal da arquitetura de coleta de métricas, explicada na figura a seguir:

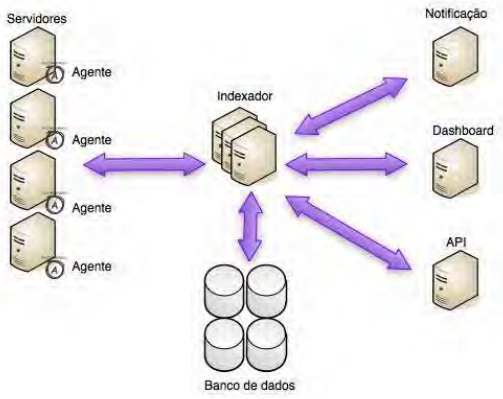


Figura 7.4: Arquitetura de coleta de métricas

Os elementos desta arquitetura podem estar na mesma máquina ou divididos em clusters para alta disponibilidade e poder de processamento. Este modo de envio de métricas se chama *push* e conta com um agente que envia métricas a um repositório, por um

indexador.

As métricas são armazenadas como *timeseries* (series de tempo). Este conceito é importantíssimo na coleta de métricas pois, na maioria das análises, uma de suas variáveis será o tempo (requests por segundo, KB/s, visitas por hora, queries por segundo). Todos os sistemas de armazenamento de métricas oferecem operações primitivas para a manipulação de timeseries.

O indexador pode ser integrado ao banco de dados ou separado. Módulos comuns ligados a ele são APIs para integrações externas, sistemas de notificação (e-mail, SMS) e dashboards. Estes sistemas podem ser adaptados de praticamente qualquer infraestrutura que você já tenha e que colete métricas. O melhor sistema de coleta de métricas é o que lhe atende e que não consome tempo excessivo para administração. O que aprendemos com o Ansible pode ajudar a criar máquinas que já entrem em produção monitoradas e coletando um conjunto mínimo de métricas.

O mesmo princípio pode ser usado para coletar métricas de dentro de suas aplicações para combinar com as do sistema operacional (tempo de consulta em banco de dados, tempo de respostas para requests HTTP, numero de clientes autenticados).

Se estes dados forem expostos por uma rota HTTP que é consultada de tempos em tempos (*polling*) ou enviados diretamente ao repositório de métricas (*push*), podem ser exibidas no mesmo dashboard ou combinadas no mesmo gráfico (por exemplo, consumo de CPU *vs.* usuários autenticados). Além de coleta de métrica, este método tem função de rastreamento de latência.

7.3 PROFILING

A última parte deste capítulo vai apresentar ferramentas de

profiling. No contexto de sistemas em produção, trata-se da coleta de métricas do ambiente de runtime e da aplicação. Estas informações são úteis para entender caminhos de código que podem ser otimizados, o efeito de aumento de latência em rede ou banco de dados para o cliente e para capturar erros que não aparecem em testes de integração.

Para a plataforma Java por exemplo, a interface *JMX* fornece muitos contadores e dados para profiling. Interpretadores de linguagens dinâmicas como Ruby, Python e PHP têm pontos de interceptação que são usados para coleta. Até mesmo o browser tem estes pontos e pode ser monitorado para erros e performances.

Vamos ver três ferramentas SaaS famosas e comerciais, mas com um nível de entrada sem custo, para introduzir este conceito: NewRelic (http://newrelic.com/), AirBrake (https://airbrake.io/) e Boundary (http://www.boundary.com/).

Elas trabalham com agentes locais e têm instruções específicas para cada plataforma. O New Relic é um profile de linguagem e introduz uma monitoração de usuário final para aplicações web chamada de *RUM* (*Real User Monitoring*). Este tipo de monitoração coleta dados dos browsers dos usuários e acumula para inferir índices de qualidade, tempo de carga, erros que o usuário vê. Em sua interface, os tempos de aplicação, redes e banco de dados são exibidos separadamente e acumulados, além da navegação por métodos e funções mais executadas e que tomam mais tempo de CPU.

O Airbrake captura exceções do código em várias linguagens em compila um índice e dashboard para consulta rápida. Em linguagens com ambiente runtime dinâmico e rico, as exceções que só aparecem em tempo de execução podem ficar em logs sem a devida atenção. Mesmo com carga sintética de testes, é difícil exercitar todos os caminhos de código e todas as situações que ambientes de

produção colocam nas aplicações.

Para monitorar e conhecer sua arquitetura, a Boundary oferece agentes que coletam *flows* (fluxos de comunicação de rede) e catalogam em um sistema que permite identificar qual o fluxo e quantidade de dados que trafega entre os elementos do seu sistema.

7.4 CONCLUSÃO

Este assunto rende um livro por si só e o nosso objetivo é criar uma estrutura minimamente aceitável e automatizada para que nenhuma máquina entre em produção sem a monitoração mínima.

Monitorar e conhecer o que acontece com sua aplicação e infraestrutura é um habito saudável. Em alto nível, as métricas que importam para o negócio podem ser derivadas de métricas ligadas diretamente à tecnologia.

Portanto, o exercício de pensar em métricas do tipo "Novos clientes por dia, paginas visitadas, cartões de créditos aprovados e vendas" ajuda a entender como combinar métricas de baixo nível para criar um histórico e avaliar qual o impacto de mudanças e incidentes no seu resultado financeiro e de satisfação do cliente.

Capítulo 8

ANÁLISE DE PERFORMANCE EM CLOUD COM NEW RELIC

Este capítulo é um exercício que usa o que você aprendeu nos capítulos anteriores para montar um ambiente na AWS com WordPress usando duas máquinas e analisar métricas coletadas sobre sua performance.

Neste ambiente, vamos instalar o New Relic e gerar uma carga sintética para entender as métricas que esta ferramenta produz. Em seguida, vamos simular intermitência entre a máquina de aplicação e a de banco de dados para avaliar qual o impacto nas métricas coletadas.

O exercício que vamos fazer é parte do dia a dia de quem administra aplicações em cloud. As máquinas virtuais nem sempre têm performance e conectividade constantes, naturalmente variam de acordo com a carga e recursos distribuídos pela máquina física que as hospeda.

Cada máquina virtual divide o tempo de processamento e recursos de I/O com seus vizinhos. Isso deu origem a um termo chamado de *Noisy Neighbour*, que explica por que máquinas de tamanho semelhante têm comportamento diferente.

Se sua aplicação é simples de ser provisionada, pode ser uma boa

prática monitorar a performance das máquinas e as recriar para que caiam em outro host, evitando que uma máquina com performance ruim degrade a qualidade de serviço.

Ter suas aplicações monitoradas e métricas bem definidas não serve apenas para saber quando o serviço está no ar e não ser pego de surpresa em caso de problemas.

Instrumentar sua aplicação para coletar métricas ajuda a ser proativo na identificação de futuros problemas e fornece dados importantes para novas versões da sua aplicação. Além disso, ajuda a manter os custos e perfis de uso de sua aplicação controlados.

8.1 WORDPRESS NA AWS

Para este capítulo, eu adaptei o playbook de instalação de WordPress com um banco de dados MySQL para ser instalado na AWS. A instalação é parecida com a do Cassandra, vamos criar duas máquinas virtuais. Estas máquinas vão se comunicar pela rede do VPC da AWS. Para facilitar a localização das máquinas, adicionei uma task para criação de tags. A chave da tag se chama `wordpress` e o valor é o tipo da máquina: `db` ou `web`.

Execute o playbook:

```
$ ansible-playbook -i aws_hosts.ini wordpress_aws.yml
--private-key ~/.ssh/aws_devel.pem
```

Se houver algum problema com a criação, revisite os passos do capítulo sobre Cassandra e suas configurações da AWS tais como VPC, chave SSH e tokens de autorização. Após a criação, configure o WordPress normalmente acessando o IP da máquina `web`. Se tiver dúvidas sobre qual máquina é a `web`, olhe no log de execução do Ansible.

Conecte-se à máquina com o comando

```
$ ssh ubuntu@ip_da_maquina -i ~/.ssh/aws_devel.pem
```

A chave usada deve ser a mesma configurada no arquivo `all` do diretório `group_vars` . Após a configuração do WordPress, faça um post de teste. Se tudo estiver certo, vamos instalar o New Relic. Não automatizei estes passos pois quero que o leitor tenha a experiência desta configuração.

8.2 INSTALANDO O NEW RELIC

Crie uma conta no New Relic (https://newrelic.com), selecione o tipo de produto para ser configurado como APM e, na página seguinte, selecione PHP. Vamos instalar o NewRelic na máquina web do nosso WordPress.

Siga as instruções de setup para PHP, aqui reproduzidas:

```
$ sudo wget -O - https://download.newrelic.com/        .gpg |
sudo apt-key add -
$ sudo echo "deb http://apt.newrelic.com/debian/
newrelic non-free" > /etc/apt/sources.list.d/newrelic.list
$ sudo apt-get update
$ sudo apt-get install newrelic-php5
$ sudo newrelic-install install
```

Após a instalação do pacote `newrelic-php5` , surgirá uma tela pedindo a chave de licença (*License Key*) que está acessível pela sua configuração do New Relic e um nome para esta aplicação. A chave aparece na página em que você escolhe qual plataforma vai configurar. Mais informações podem ser encontradas na página do agente PHP (https://docs.newrelic.com/docs/agents/php-agent/installation/php-agent-installation-ubuntu-debian).

A configuração pode aparecer com uma tela pedindo apenas a licença ou pode ser um sistema de menu, depende do momento da instalação. Em caso de dúvida execute `newrelic-install` e siga as instruções.

Reinicie o *nginx* e o PHP5-FPM:

```
$ /etc/init.d/nginx restart
$ /etc/init.d/php5-fpm restart
```

Volte à página do New Relic e verifique se os dados de sua aplicação já chegaram. Em caso de dúvida, verifique se o processo *newrelic-daemon* está sendo executado:

```
$ sudo ps -ef | grep newrelic
```

Se nenhum processo aparecer, execute:

```
$ sudo newrelic-daemon
```

Examine os logs do `newrelic` em:

```
$ tail -f /var/log/newrelic/*
```

Navegue um pouco no blog, crie um post e, de volta ao site do New Relic, clique na opção `Browser` e selecione o nome de sua aplicação. Deve aparecer uma tela como a seguir:

Figura 8.1: New Relic

Dentro da máquina `web` execute o comando:

```
$ sudo tcpdump -i eth0 | grep -i newrelic
```

O comando `tcpdump` coloca a interface em modo promíscuo —

todos os pacotes de rede que passam por ela são interceptados e filtrados de acordo com a regra passada ao comando.

A regra que usamos é simples, só define a interface. O filtro usado em `grep -i newrelic` serve para isolar linhas que tenham esta string. O que vemos em seguida são os pacotes enviados da máquina local ao coletor de dados do New Relic, como no exemplo a seguir:

Nosso blog não é famoso, mas vamos simular uma certa carga para entender, por variáveis do sistema operacional, o que acontece em um momento de carga.

8.3 GERANDO CARGA — PROFILING DA APLICAÇÃO

Existem muitos contadores e ferramentas em ambiente Linux para *profiling* de sistema operacional e aplicações. Profiling é um conjunto de técnicas que ajuda a diagnosticar problemas e comportamentos do sistema operacional e da aplicação.

A observabilidade é importante antes e no momento de algum problema. Antes, serve parar criar o que chamamos de *baseline*, o comportamento padrão do conjunto e desvios perigosos que podem gerar indisponibilidades. No momento, servem para encontrar problemas e pontos de contenção.

A maioria dos recursos de um sistema operacional é modelada como filas. As filas dependem de um ou mais produtores e consumidores. Em um momento de alta requisição, estas filas podem crescer e provocar lentidões. Além disso, o acumulo de itens ou de pedidos para itens pode gerar o colapso do sistema operacional. Os principais ofensores são dispositivos de disco, rede, CPU e memória.

Entre paradigmas de concorrência, problemas de código e sobrecarga, as causas e sintomas podem se combinar. O New Relic permite uma visão sistêmica da aplicação dividida entre elementos que combinam requisitos distintos de CPU, memória e I/O.

Em nossa máquina local ou em uma máquina virtual fora deste cluster, vamos executar o comando a seguir para gerar uma carga previsível:

```
$ while true; do ab -c 10 -n 100 http://  . .   .   /; sleep 5;
done
```

Este comando usa o `ab` (*apache benchmark*) para enviar 100 requests, com concorrência de 10 requests por vez de 5 em 5 segundos. Não é uma carga realista nem possui um crescimento rápido, mas é uma carga sintética que ajudará a entender os elementos apresentados no New Relic.

A primeira tela que vamos examinar é da sessão *APM*, chamada *Overview*:

Figura 8.2: NewRelic Web Transactions

Esta tela tem o resumo do tempo de resposta das transações web. A curva tem início no momento em que a geração de carga sintética começou.

Ao lado, temos o Apdex Score — um índice padrão da indústria para descrever a qualidade que o usuário final está vivenciando, um pequeno gráfico de *throughput* que usa a métrica RPM (*requests per minute*) para descrever pedidos de clientes para a aplicação. Descendo a tela, teremos uma lista de transações descrita pelas páginas e rotas de API chamadas.

A próxima tela que veremos é a sessão *databases* do *APM*:

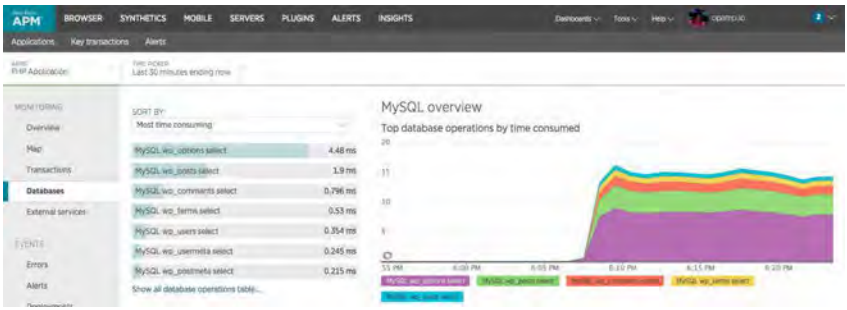


Figura 8.3: NewRelic DB Transactions

O gráfico principal é o acumulado de todas as transações de banco de dados recuperadas pelo New Relic. Lembre-se que não instalamos nada na máquina `db` mas o New Relic consegue recuperar estas transações pelo PHP.

A ordem das queries é da mais lenta para a mais rápida. Neste regime de carga, os tempos são bem baixos, mas podemos ver um padrão: a query `wp_options select` é a que ocupa mais tempo neste gráfico. O menu pull-down acima da lista de queries com a legenda *Sort-by* oferece outros critérios de ordenação.

Vamos aumentar o número de transações do teste de carga:

```
$ while true; do ab -c 20 -n 200 http://  . .   .   /; sleep 5;
done
```

Dobramos a concorrência e o número de requests, mantivemos o tempo entre os lotes de requests. A tela de *databases* ficou assim:

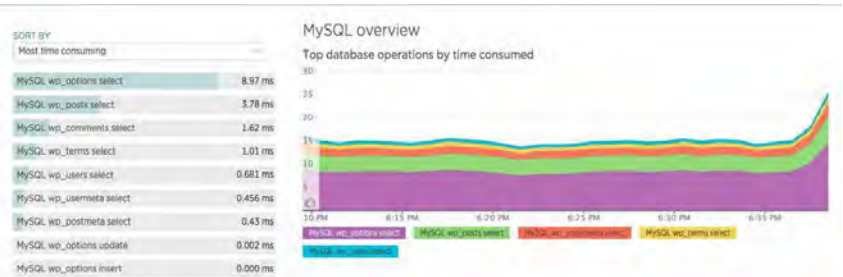


Figura 8.4: NewRelic DB Transactions

Os tempos das queries dobraram, o que pode dar a ideia de que o consumo de recursos é linear. A prática ensina que nem sempre é assim, que a partir de um número de requests por segundo alguns recursos esgotam antes de outros.

Abaixo do gráfico de tempo de operações, há o gráfico de tempo de resposta do banco de dados:

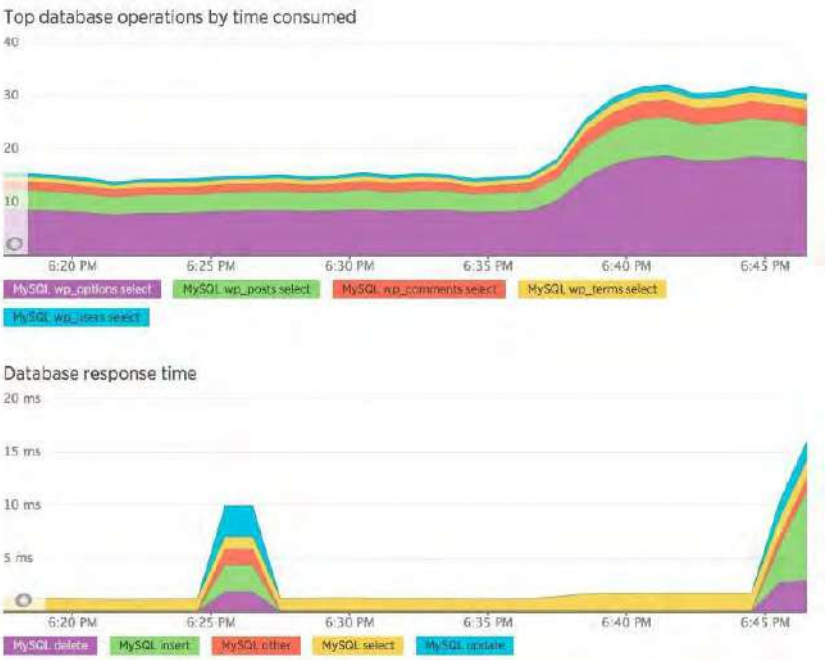


Figura 8.5: NewRelic DB Response time

Os dois picos foram posts que criei no blog. Para o padrão de acesso do teste de carga, a operação predominante é o `SELECT`. Para um novo post, quase todas as operações são utilizadas.

Os tempos de resposta mudam por conta disso. Se olharmos o nosso teste de carga, uma rodada normal tem a seguinte distribuição de percentuais na configuração de máquinas que utilizamos:

```
Percentage of the requests served within a certain time (ms)
```

Durante um post, a distribuição muda para :

```
Percentage of the requests served within a certain time (ms)
```

De 90% em diante, os tempos aumentam significantemente. Em 99%, o tempo aumenta em quase 1 segundo. O consumo de recursos foi afetado por uma operação diferente da operação utilizada no teste de carga. Um teste de carga completo deveria simular usuários postando mensagens de forma concorrente.

Dentro da máquina `web` , vamos ver a carga (`load avg`) e estatísticas de CPU.

```
root@ip-10-0-0-100:/etc/php5/fpm# uptime


root@ip-10-0-0-100:/etc/php5/fpm# vmstat
procs -----------memory---------- ---swap-- -----io----
 r  b   swpd   free   buff  cache   si   so    bi    bo


 -system-- ------cpu-----
   in   cs us sy id wa st
```

Vamos aumentar a carga:

```
$ while true; do ab -c 40 -n 400 http://  . .   .   /; sleep 5; do
ne
```

A saída dos comandos mudou um pouco:

```
root@ip-10-0-0-100:/etc/php5/fpm# uptime


root@ip-10-0-0-100:/etc/php5/fpm# vmstat
```

```
procs -----------memory---------- ---swap-- -----io----
 r  b   swpd   free   buff  cache   si   so    bi    bo


 -system-- ------cpu-----
 in   cs us sy id wa st
```

A carga aumentou e o consumo de memória também, mas nada agressivo. Nas duas medições, a última coluna do bloco CPU, chamada `st`, está com valor 1. Esta coluna representa o tempo roubado (*stolen time*) pelo hypervisor. É um indicador de que a máquina virtual não tem o tempo que esperava para executar. É um contador importante para ser monitorado. Caso o tempo roubado cresça ou fique constante acima de 0, é hora de recriar a máquina virtual em outro host. Com a automação que construímos isso é fácil.

Para nosso teste esta é uma dificuldade interessante. No New Relic, esta é a cara do consumo de recursos:

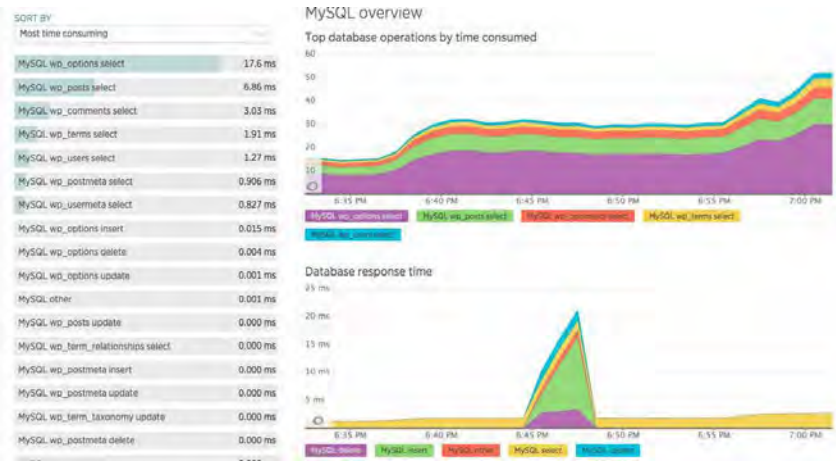


Figura 8.6: NewRelic DB Response time

Os tempos aumentaram e durante o teste minha conexão SSH travou. Como o perfil de máquina que escolhi é pequeno, este

comportamento não é raro.

O teste que quero fazer agora é reduzir a banda da interface da máquina de banco de dados para gerar filas em ambos os lados. Conecte-se à maquina *db* e vamos ver como estão a carga e os contadores de CPU:

```
ubuntu@ip-10-0-0-242:~$ uptime


ubuntu@ip-10-0-0-242:~$ vmstat
procs -----------memory---------- ---swap-- -----io----
 r  b   swpd   free   buff  cache   si   so    bi    bo


 -system-- ------cpu-----
 in   cs us sy id wa st
```

8.4 SIMULANDO LIMITAÇÕES DE REDE

Vamos usar a documentação do Linux Advanced Routing How-To (http://lartc.org/howto/lartc.ratelimit.single.html) para limitar a banda desta máquina em 256kbps.

```
root@ip-10-0-0-242:~# export DEV=eth0
root@ip-10-0-0-242:~# tc qdisc add dev $DEV root handle 1:
cbq avpkt 1000 bandwidth 100mbit
root@ip-10-0-0-242:~# tc class add dev $DEV parent 1:
classid 1:1 cbq rate 256kbit allot 1500 prio 5 bounded isolated
root@ip-10-0-0-242:~# tc filter add dev $DEV parent 1:
protocol ip prio 16 u32 match ip dst 10.0.0.100 flowid 1:1
```

A primeira mudança que vemos é nos percentuais do teste de carga. Antes da limitação da interface:

```
Percentage of the requests served within a certain time (ms)
```

Após a limitação, o teste demora mais para finalizar. Estes tempos tendem a aumentar conforme mais requests são lançados no banco de dados.

```
Percentage of the requests served within a certain time (ms)
```

Os números da máquina de banco de dados continuam bons:

```
root@ip-10-0-0-242:~# uptime


root@ip-10-0-0-242:~# vmstat
procs -----------memory---------- ---swap-- -----io----
 r  b   swpd   free   buff  cache   si   so    bi    bo


 -system-- ------cpu-----
  in   cs us sy id wa st
```

Na máquina `web`:

```
ubuntu@ip-10-0-0-100:~$ uptime


ubuntu@ip-10-0-0-100:~$ vmstat
procs -----------memory---------- ---swap-- -----io----
 r  b   swpd   free   buff  cache   si   so    bi    bo


-system-- ------cpu-----
 in   cs us sy id wa st
```

O *loadavg* melhorou e o *stolen time* está alto. Vamos olhar o New Relic:

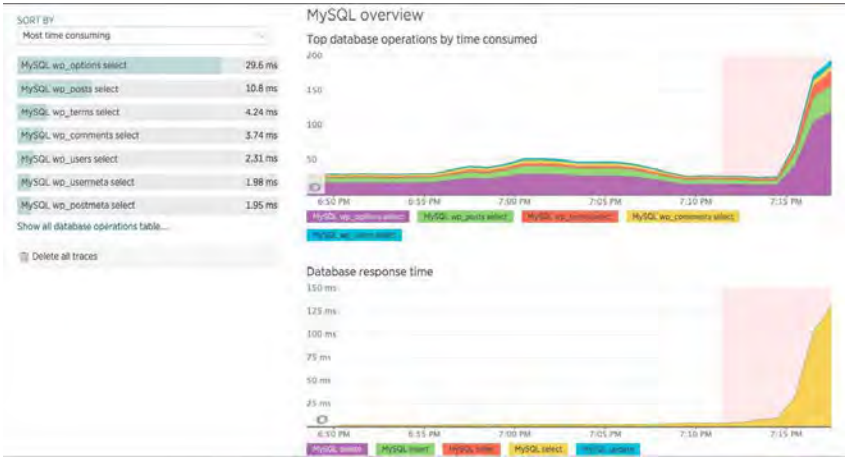


Figura 8.7: NewRelic DB Response time

A maior inflexão de tempo aconteceu no momento em que a limitação foi instalada na interface da máquina de banco de dados. Vamos remover a limitação e verificar os tempos.

```
root@ip-10-0-0-242:~# tc qdisc del dev $DEV root
```

Após alguns minutos, vemos os percentuais do teste voltarem aos patamares anteriores:

```
Percentage of the requests served within a certain time (ms)
```

Os contadores da máquina mudaram. Por conta da vazão que o servidor tem, o *load average* aumentou. O teste de carga é executado

mais rápido, portanto executa mais vezes.

```
ubuntu@ip-10-0-0-100:~$ uptime


ubuntu@ip-10-0-0-100:~$ vmstat
procs -----------memory---------- ---swap-- -----io----
 r  b   swpd   free   buff  cache   si   so    bi    bo


-system-- ------cpu-----
 in   cs us sy id wa st
```

Se olharmos no New Relic:

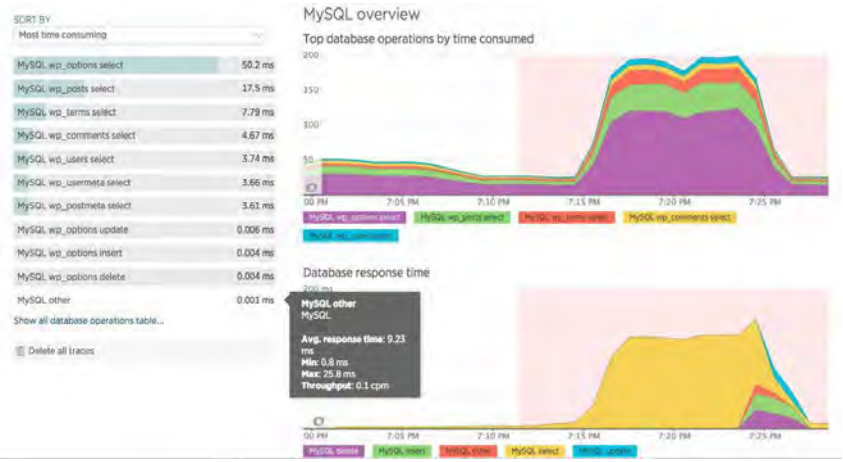


Figura 8.8: NewRelic DB Response time

O impacto da redução de banda da interface da máquina de banco de dados foi significativo. Se o teste simulasse um fluxo normal de clientes, haveria mais falhas do lado do cliente.

Estamos falando de um blog simples sem acesso respondendo à página em 15s em testes com 400 conexões distribuídas em lotes de 40 conexões simultâneas, que subiu para 36 segundos no pior momento do teste:

```
Percentage of the requests served within a certain time (ms)
```

Este número alto é efeito do tempo roubado e da limitação de rede no MySql. Vamos ver o gráfico do *Overview*:

Figura 8.9: NewRelic Web Transactions

A história que ele conta é que a carga foi aumentando um pouco antes de 7:10PM, e que a composição entre PHP e MySQL mudou quando foi instalada uma restrição de 256kbps (*kbits per second*). Os tempos subiram mas a aplicação PHP e sua máquina pareciam mais tranquilas.

Esta impressão se deve ao fato de que cada requisição demorava mais e vinha da mesma origem. O efeito é contrário ao que estava acontecendo. Se olhássemos apenas o load, teríamos a impressão de que restringir a velocidade foi uma melhoria. Só que, olhando para o relatório do teste, os tempos agrupados por percentuais dos requests aumentava dramaticamente.

O New Relic tem um módulo chamado *Browser* que mede alguns dados relacionados ao frontend. O *AB* não simula um browser completo, portanto a coleta de dados do New Relic fica prejudicada. Mas os dados dos posts foram colhidos e o do APDex calculado. Desde o inicio do teste, eu fiz alguns posts no WordPress e, após desligar o teste, fiz outro post para analisarmos a recuperação da aplicação.

Como algumas instâncias da AWS tentam fazer *bursting*, a alta carga imposta pelo teste solicitou mais capacidade, que não estava disponível. Para este volume do teste a instância não era o melhor tipo, mas foi uma limitação que escolhi para demonstrar estes contadores e efeitos. Ao finalizar o teste, os contadores da máquina voltaram ao normal:

```
ubuntu@ip-10-0-0-100:~$ uptime


ubuntu@ip-10-0-0-100:~$ vmstat
procs -----------memory---------- ---swap-- -----io---- -system--
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs


 ------cpu-----
  us sy id wa st
```

A primeira medida do VMSTAT não tem com o que comparar, portanto vale como uma medida tópica mas deve ser observada ao longo do tempo.

Após alguns minutos coletando, vamos ver o New Relic:

Figura 8.10: NewRelic Browser

Selecionei um espaço de 3 horas para mostrar como os posts anteriores ao teste ficaram quase nulos perto do resultado do sistema sob estresse. O APDex também caiu rapidamente. Voltando ao APM, vamos olhar o item *Map*:

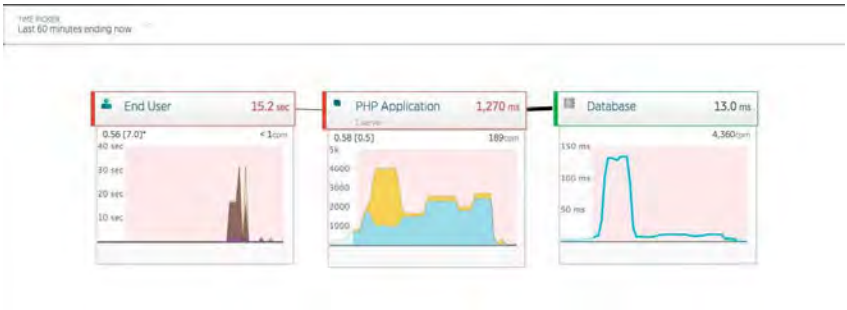


Figura 8.11: NewRelic Browser

O crescimento de tempo de resposta da aplicação coincide com o do banco de dados. A janela é a da restrição da interface mas o impacto de 100ms no banco é multiplicado na camada de aplicação.

8.5 CONCLUSÃO

Este caso que criamos assumiu algumas simplificações: fonte única de trafego, constância no volume de requisições sem curva de

crescimento e um setup simples de duas máquinas. Ele serviu para demonstrar algumas ferramentas e uma metodologia simples para simular erros comuns.

Uma aplicação rodando em cloud está sujeita a variações de performance que podem gerar resultados imprevisíveis. Um sistema operacional que tem velocidade variável de rede e disco pode mudar o comportamento de uma aplicação tradicional desenvolvida em máquina física. Além das variações de ambiente, as variações de carga podem criar sintomas que nos afastam da causa raiz.

Eu recomendo ler o livro Systems Performance: Enterprise and the Cloud (http://www.brendangregg.com/books.html) do Brendan Gregg. Para cada elemento do sistema operacional ele propõe ferramentas e métodos para descobrir pontos que podem ser melhorados.

Neste capítulo, abordei sob a perspectiva de uma aplicação web simples mas existem casos complexos que envolvem bancos de dados e sistemas assíncronos e não são triviais.

A experiência conta e a melhor maneira é começar como no capítulo anterior: criando monitoração e métricas para suas aplicações.

Capítulo 9

DOCKER

Neste capítulo vamos ver uma outra categoria de virtualização chamada *Operating system virtualization*.

A virtualização mais conhecida envolve a emulação de uma máquina completa, controlada por um agendador de tarefas que divide o tempo e recursos do host entre múltiplas instâncias chamadas de máquinas virtuais.

Preparei um playbook do Ansible que cria uma máquina virtual com o kernel e aplicações necessárias para executar *containers*. Ela também tem o *Docker* instalado, que é uma API e daemon que implementa uma forma de utilizar containers.

9.1 VIRTUALIZAÇÃO

Todo tipo de virtualização lida com a divisão de recursos e o isolamento dos *tenants* (usuários). Ao emular uma máquina completa, o host entrega uma interface que simula o *bare-metal*, termo utilizado para descrever o hardware, com BIOS, drivers e espaço de memória. Na virtualização tradicional podemos ter *guests* (maquinas virtuais) dos tipos PV e HVM.

PV é a sigla de *paravirtualization*, na qual a CPU do host não precisa ter extensões específicas de virtualização e depende inteiramente do suporte do kernel do sistema operacional do host e do guest para emular e acessar dispositivos. Esta forma de

virtualização permite boa performance e significa que os guests conhecem o host.

HVM significa que o host utiliza extensões e camadas de software para emular dispositivos. A mais famosa se chama QEMU e é utilizada em vários *hypervisors* (software utilizado para gerenciar máquinas virtuais).

Entre estes dois modelos, existem combinações de drivers, acessos e modelos de performance.

A categoria de *Operating system virtualization* engloba sistemas que isolam mais de uma instância de *userspace*. Esta modalidade existe há algum tempo sob diferentes nomes e recursos em diferentes tipos de Unices (plural de Unix) e no Windows.

Aquela em que vamos nos concentrar está presente no Linux e se chama *containers*. Junto aos containers, para gerenciar os userspaces é utilizado *CGroups*.

A diferença de performance e *overhead* (custo) ao sistema operacional é significativa. Como vamos confirmar, executar um container é mais rápido do que bootar uma máquina virtual.

Isso acontece pois em máquinas virtuais o sistema operacional recebe a abstração de hardware que, implicitamente, tem camadas que devem ser gerenciadas, e o container é isolado por abstrações de namespace em um mesmo sistema operacional.

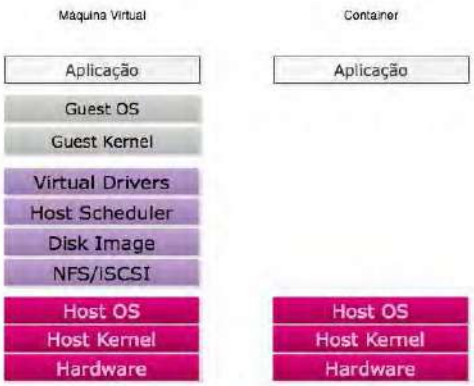


Figura 9.1: VM e containers

Máquinas virtuais e containers não simples de comparar. A figura anterior é uma descrição simples de camadas entre a aplicação e os recursos de hardware mas existem mais controles complexos em ambos os lados. Uma tecnologia não anula a outra pois os avanços na tecnologia de virtualização nos permitem dividir máquinas que um sistema operacional não seria tão eficiente em dividir. São duas aplicações de *timeshare* e de isolamento computacional importantes.

9.2 CONTAINERS E CGROUPS

CGroups é a implementação do Linux de namespaces. Junto com as ferramentas e drivers de containers, compõe a infraestrutura para containers (https://linuxcontainers.org/) do kernel. LXC é o conjunto de ferramentas para manipular containers diretamente.

O CGroups não é útil apenas para containers. Você pode criar grupos para limitar o consumo de memória de aplicações em sua

máquina virtual.

Se aplicarmos uma regra para limitar o consumo de memória de um grupo a 70%, todos os processos deste grupo serão avaliados por esta regra. Quando o grupo ultrapassa o limite preestabelecido de memória, o kernel tenta recuperar memória não alocada dos processos. Se ainda assim o consumo estiver alto, ele usa o OOM-Killer para matar o maior processo do grupo.

A arquitetura e detalhes do CGroups é densa e pode ser lida na documentação do kernel (https://www.kernel.org/doc/Documentation/cgroups/memory.txt). Basta dizer que ele faz o mesmo controle para I/O, rede e CPU e que foi reescrito nas últimas versões do kernel. Sua estrutura é a base de controle de sistemas que controlam outros recursos além dos containers.

Os containers se aproveitam desta e de outras características de controle do kernel do Linux para implementar o isolamento e instanciação de userspaces distintos.

O container não é a máquina virtual completa, ele é um *root filesystem* (imagem de um filesystem) que o kernel usa para isolar um usuário.

As imagens que o comando `lxc-create` utiliza são mínimas, contêm o necessário para usar o sistema operacional sem um kernel. As possibilidades são interessantes, já que você pode criar uma imagem especializada e executar em um container. Esta imagem pode conter sua aplicação e bibliotecas necessárias para a execução.

Existem diferentes configurações de redes que um container pode usar. Vamos nos limitar à configuração padrão, que é uma bridge apresentada ao container que se liga à interface principal do kernel.

Vamos criar um container com o sistema operacional Ubuntu para ver como funciona na prática. No diretório `ansible-docker-bootstrap` digite `vagrant up` e espere a máquina subir.

Em seguida, execute `vagrant ssh`. Certifique-se de que está na máquina virtual com o comando `*uname -a` e `ifconfig -a`. Depois, digite os comandos a seguir:

```
$ sudo -s
# apt-get install lxc
# lxc-create --name teste -t ubuntu
```

Após alguns minutos, o container estará instalado e em execução. O próximo container deste tipo usará os arquivos foram baixados na primeira instalação. Podemos verificá-los em execução:

```
# lxc-ls
teste
```

O único container listado foi o `teste` que criamos. Vamos executá-lo:

```
# lxc-start --name teste
```

Após o boot, digite `enter`. Você vai cair em um prompt pedindo usuário e senha. Use `ubuntu/ubuntu` e explore o container. Verifique a interface de rede com `ifconfig -a` e saia do com o comando `sudo halt`. Este comando vai desligar o container. Vamos executá-lo agora em backgroud:

```
# lxc-start --daemon --name teste
# lxc-console --name teste
```

Você vai cair novamente no prompt de usuário e senha. Para sair do container, use a combinação de teclas `CONTROL-A-Q`. Conecte novamente no container com `lxc-console` e execute o comando a seguir:

```
$ vim oie
```

Saia com `CONTROL-A-Q` e, no host, execute o comando:

```
# ps -ef | grep vim
```

```
--color=auto vim
```

A saída mostra o comando `vim oie` que foi executado dentro do container. Os PIDs podem mudar, mas a saída é a mesma. Volte ao container com `lxc-console` . Pode digitar `ESC-:q` , pois a tela do Vim pode não voltar corretamente.

Digite `top` e veja se consegue ver algum processo do host. Este é o principio do isolamento. Dá máquina host você consegue ver os processos dos containers, mas do containers e entre containers só consegue ver o que o isolamento permite. Vamos parar o container:

```
# lxc-stop --name teste
```

Existe muito mais que pode ser feito com containers sem outras ferramentas. O conceito é poderoso e maduro, com blocos bem definidos. Sistemas como Hadoop, Mesos, Google Kubernetes e Docker utilizam estes princípios para construir aplicações modulares. Vamos nos aprofundar no Docker e seu ecossistema no resto deste capítulo.

9.3 DOCKER

O Docker é um projeto de código aberto que define um fluxo de trabalho e integração para os recursos de containers, namespaces, grupos e redes. Este fluxo é apresentado por uma API e um programa de linha de comando que usa esta API para gerenciar o ciclo de vida de containers.

Além disso, o projeto fornece uma linguagem para descrever o container e repositórios para que as imagens criadas sejam versionadas. Uma vez criada a imagem, um ou mais containers

podem ser executados com ela, criando múltiplas instâncias na mesma máquina.

Vamos usar um exemplo de servidor HTTP em Python para demonstrar a criação de uma imagem e a execução em uma máquina virtual.

A máquina virtual que vamos utilizar é a mesma em que testamos o LXC. Após executá-la com `vagrant up` e conectar com `vagrant ssh`, vá ao diretório `/vagrant/dockerapp`. Dentro dele, temos dois arquivos: `Dockerfile` e `http_server.py`.

`Dockerfile` é o nome do arquivo utilizado pelo *Docker* para construir uma imagem nova de um container. A organização lembra um playbook de Ansible. Vamos examinar o `Dockerfile` do nosso teste para entender um pouco mais:

```
FROM ubuntu:14.04
RUN apt-get -y install python
ADD http_server.py /http_server.py
CMD ["python", "/http_server.py"]
```

A diretiva `FROM` indica qual imagem será utilizada como base. Pode ser tanto uma de sistema operacional quanto uma já configurada de alguma aplicação.

Uma das convenções poderosas que o Docker introduziu foi a ideia de repositórios de imagens com versionamento. A cada comando no `Dockerfile`, uma nova versão local da imagem é criada.

Esta versão é uma camada em cima do sistema de arquivos desta imagem. Como os containers, as imagens do Docker só contêm arquivos, elas não têm *kernel* e *drivers* — são o que chamamos de *Root Filesystem*.

Todo sistema operacional monta uma imagem que vive no disco

quando inicializa. Esta imagem pode estar escrita diretamente nos blocos do disco ou em um arquivo que vive dentro de outro *filesystem* no disco. O Docker tem ferramentas que facilitam a criação incremental e o versionamento destes arquivos.

Após escolher o sistema operacional Ubuntu 14.04 usei a diretiva *RUN* para instalar o Python e *ADD* para adicionar um arquivo local a um caminho especificado. Por fim utilizei *CMD* para executar o meu programa.

Outro conceito fundamental do Docker é que cada container executa um processo apenas. O princípio de responsabilidade é importante: containers não são máquinas virtuais e a convenção utilizada pelo Docker é de que eles não têm nem acesso SSH.

Este princípio e a facilidade de criar e instanciar containers permitem a composição e agregação de containers em arquiteturas complexas, que podem ser reaproveitadas. O nome em inglês deste principio é *composability* na documentação do Docker. Adiante veremos uma ferramenta para orquestrar mais de um container e ligá-los para compor a arquitetura de uma aplicação web.

Para utilizar um container, temos que fazer o *build*. O build faz downloads da imagem e das camadas que a compõem para o repositório local.

Durante o , o Docker vai emitir vários números precedidos de `--->` , que são IDs de operações do Docker. Estes números são a referência para as camadas e para a imagem final. Vamos usar o comando `build` para criar nossa imagem, dentro do diretório `dockerapp`:

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker build .
Sending build context to Docker daemon 4.096 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04
14.04: Pulling from ubuntu
```

```
ubuntu:14.04: The image you are pulling has been verified.
Important: image verification is a tech preview feature and
should not be relied on to provide security.



Status: Downloaded newer image for ubuntu:14.04

Step 1 : RUN apt-get -y install python

Reading package lists...
Building dependency tree...
Reading state information...
The following extra packages will be installed:
  libpython-stdlib libpython2.7-minimal libpython2.7-stdlib
  python-minimal
  python2.7 python2.7-minimal
Suggested packages:
  python-doc python-tk python2.7-doc binutils binfmt-support
The following NEW packages will be installed:
  libpython-stdlib libpython2.7-minimal libpython2.7-stdlib
  python python-minimal python2.7 python2.7-minimal
0 upgraded, 7 newly installed, 0 to remove and 0 not upgraded.
Need to get 3734 kB of archives.
```

Após a atualização dos repositórios e a instalação dos pacotes:

```
Setting up python-minimal (2.7.5-5ubuntu3) ...
Setting up python (2.7.5-5ubuntu3) ...


Step 2 : ADD http_server.py /http_server.py


Step 3 : CMD python /http_server.py
```

Todos os passos (*Step*) foram executados com sucesso e nossa imagem foi criada localmente com o ID . O Docker

provê abstrações parecidas com as do sistema operacional para gerenciar instâncias de containers. Uma delas é o comando `ps` :

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker ps
CONTAINER ID        IMAGE    COMMAND    CREATED    STATUS
PORTS               NAMES
```

Não temos nenhum container sendo executado. As colunas indicam um ID de *execução*, o ID da imagem, qual comando este container está executando, status, portas de rede expostas e um nome temporário. As operações sobre este container em execução devem ser feitas utilizando o `CONTAINER ID` ou o `NAME` dado a ele, não o `ID` ou `NAME` da imagem original.

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker run

^CTraceback (most recent call last):
  File "/http_server.py", line 39, in <module>
    ReuseAddrTCPServer(("", PORT), ServerHandler).serve_forever()
  File "/usr/lib/python2.7/SocketServer.py", line 236, in
    serve_forever poll_interval)
  File "/usr/lib/python2.7/SocketServer.py", line 155,
    in _eintr_retry
    return func(*args)
KeyboardInterrupt
```

O container pode ser executado em *foreground* ou em *background* ou *detached*. Na chamada anterior, executei em foreground e o terminal ficou preso a ele. Usando `CONTROL+C` eu abortei a execução e vou tentar novamente com a chave `-d` (*detach*):

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker run -d


root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker ps
CONTAINER ID        IMAGE                 COMMAND
PORTS               NAMES
                                :latest   "python /http_server

CREATED             STATUS
seconds             elated_bohr
```

O comando `docker ps` mostra a instância da imagem 2836de6b7707, versão *latest*, sendo executada. Nenhuma porta foi exposta, portanto não conseguimos acessar a aplicação. Vamos matar e executar este container da forma correta.

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker kill


root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker run -d


root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker ps
CONTAINER ID        IMAGE                 COMMAND
PORTS                       NAMES


CREATED
2 seconds ago Up 1 seconds 0.0.0.0:10000->10000/tcp

STATUS
trusting_lovelace
```

Nesta execução, usei a opção `-d` para que o container execute em background, e a opção `-p`, equivalente à diretiva *EXPOSE* que poderia ser inserida no `Dockerfile` para indicar qual porta do container quero expor e qual porta do *host* é a equivalente. Neste caso, `http_server.py` é um programa que abre a porta 10000 e eu optei por manter a mesma porta no host. A ordem é `-p porta_do_host:porta_do_container`.

Falta um teste, feito no host:

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# curl -v
localhost:10000
* Rebuilt URL to: localhost:10000/
* Hostname was NOT found in DNS cache
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 10000 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:10000
> Accept: */*
>
```

```
* Connection #0 to host localhost left intact
<html><body><form action='/' method=POST>
<textarea name=msg rows='10' cols='100'></textarea><br>
<input type='submit' name='submit'></form></body></html>
```

No momento do build, poderíamos ter usado o comando:

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker build
-t python_app .
```

A chave `-t` dá um nome arbitrário a este build que pode ser usado no lugar do ID.

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker run -d
-p 5000:10000 python_app

root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker ps
CONTAINER ID        IMAGE               COMMAND
PORTS               NAMES
                    python_app:latest   "python /http_server

CREATED       STATUS
5 seconds ago Up 4 seconds 0.0.0.0:5000->5000/tcp   high_sammet
```

Podemos inspecionar o container com o comando `inspect`, que retorna um `JSON` com todos os dados sobre a instância ou dados específicos:

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker inspect
high_sammet
[{
    "AppArmorProfile": "",
    "Args": [
        "/http_server.py"
    ],
    "Config": {
        "AttachStderr": false,
        "AttachStdin": false,
        "AttachStdout": false,
        "Cmd": [
            "python",
            "/http_server.py"
        ],

...

  "Path": "python",
```

```
    "ProcessLabel": "",
    "ResolvConfPath": "/var/lib/docker/containers/

    resolv.conf",
    "RestartCount": 0,
    "State": {
        "Dead": false,
        "Error": "",
        "ExitCode": 0,
        "FinishedAt": "000 -  -     :00:   ",
        "OOMKilled": false,
        "Paused": false,
        "Pid": 9   ,
        "Restarting": false,
        "Running": true,
        "StartedAt": "2015-  -     :  :  .          "
    },
    "Volumes": {},
    "VolumesRW": {}
}
]
```

Vamos pegar o IP interno, porta e mapeamento:

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker inspect
-f '{{ .NetworkSettings.IPAddress }}' high_sammet


root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker port
high_sammet
10000/tcp -> 0.0.0.0:5000
```

No comando `inspect`, passamos o caminho do `JSON` da mesma forma que os templates Jinja2 do Ansible. O comando `port` mostrou o mapeamento criado de origem e destino.

Nos três últimos comandos, usei o nome atribuído pelo docker a este container. Com estes dados, podemos configurar nosso *nginx* local para servir a aplicação utilizando a técnica de proxy reverso. Desta forma, poderíamos ter mais instâncias da aplicação rodando em containers e o *nginx* cuidando da comunicação com o mundo exterior.

Para ficar completo, só precisamos de uma forma de atualizar a

configuração do *nginx* toda vez que novos containers executarem ou finalizarem.

Vamos usar uma configuração estática com apenas um backend de container:

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# cp app.conf
/etc/nginx/sites-available/
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# ln -s
/etc/nginx/sites-available/app.conf /etc/nginx/sites-enabled/
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp#
/etc/init.d/nginx restart
 * Restarting nginx nginx
   ...done.
```

Abra o browser de sua máquina em http://192.168.33.2 e use a aplicação. A configuração do *nginx* espera um backend ouvindo na porta 5000, que é uma porta com *forward* da porta 10000 do container. No host, inspecione o container, os processos e logs gerados por ele.

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# docker logs
```

O nome do container deve mudar de acordo com seu sistema operacional. Ele é selecionado aleatoriamente pelo Docker. Se você olhar o log do *nginx* verá uma entrada semelhante a esta:

```
root@vagrant-ubuntu-trusty-64:/vagrant/dockerapp# tail -f
/var/log/nginx/app.access.log
```

Se você criar uma conta no Docker Hub (https://registry.hub.docker.com/), poderá usar o comando `commit`, e `push` pode colocar sua imagem no *Registry* e, dessa forma, permitir que outros usuários executem sua aplicação.

O Docker Hub é um repositório que funciona de forma semelhante ao Git. Você pode utilizá-lo para encontrar imagens de aplicações completas e com elas compor suas próprias imagens. Você pode ter um *Registry* privado também, instalado em sua infraestrutura, caso suas imagens contenham software proprietário ou dados sensíveis.

A documentação do Docker é completa e contém exemplos interessantes junto a boas práticas atualizadas. Sugiro seguir o User Guide (https://docs.docker.com/userguide/) para entender melhor o modelo de redes (Networking) e de associação (Link) dos containers.

9.4 BOOT2DOCKER

Na sessão anterior, eu usei uma máquina virtual no Vagrant para trabalhar com o Docker. Eu prefiro este caminho pois o mesmo playbook que usei para criar esta máquina virtual servirá para criar e configurar meus servidores em produção.

Mas dependendo do seu setup você pode querer executar os containers fora do seu sistema operacional e ter uma experiência parecida com executando-os localmente.

O Docker permite o acesso do cliente de linha de comando a APIs remotas. Baseado nisso foi criado o projeto *Boot2Docker*, que nada mais é do que uma máquina virtual e um conjunto de configurações para o *VirtualBox* que permite enviar comandos ao

Docker sem estar em uma sessão SSH dentro da máquina virtual.

O Boot2Docker é suportado pela empresa Docker e atualizado constantemente. Pode ser uma boa opção caso você esteja em um ambiente sem Vagrant ou queira seguir os exemplos da documentação do Docker sem se preocupar em montar a configuração da máquina virtual.

As instruções para instalação (https://docs.docker.com/installation/mac/) detalham este conceito e lembram que o Boot2Docker é um ambiente de desenvolvimento — não deve ser utilizado em produção. É uma alternativa ao método que apresentei e tem o mesmo comportamento. Em vez de executar os comandos em uma sessão SSH, você poderá executá-los no terminal do seu MAC ou Linux.

9.5 CRIANDO NOSSO CONTAINER DE PHP E NGINX E USANDO O DOCKER HUB

Além do gerenciamento de containers, o Docker tem ferramentas para uso de sua API e provisionamento de sistemas. Vamos usar o Docker Compose para provisionar um WordPress como fizemos anteriormente.

Provisionar aplicações em containers requer modificações para convenções que se aproximem do padrão 12 Factor App (http://12factor.net/).

É uma abordagem interessante para a arquitetura de aplicações e que usei durante o livro para algumas escolhas, por exemplo, proxy reverso e códigos que podem receber uma porta para ouvir tanto por configuração quanto por variáveis de ambiente.

Este item é chamado de port binding (http://12factor.net/port-binding) na documentação e é ao que vamos nos atentar para criar

um ambiente com WordPress no Docker.

A maioria das aplicações aceita uma porta para ouvir HTTP que está dentro do código (errado) ou em um arquivo de configuração (melhor do que *hardcoded*). A especificação sugere que aplicações não devem ter portas fixas pois não há localidade fixa para serem executadas.

Se levarmos isso para o contexto de containers, anteriormente tivemos que falar para o sistema operacional relacionar uma porta do host com uma porta interna do container. Todo container tem um endereço IP inválido e acessa o mundo exterior através de regras e interfaces configuradas no host.

As aplicações neste modelo são componentes que escalam pelo modelo de processos e containers, ou mesmo máquinas virtuais. Considerando que estes são recursos computacionais, poderíamos executar um componente onde houvesse capacidade computacional disponível.

Se a porta ou algum recurso for fixo ou limitado por locais onde as regras de rede existam ou *loadbalancers* que utilizam *sticky-bit sessions* (e devem saber exatamente aonde mandam cada cliente) esta flexibilidade de provisionamento fica prejudicada.

O Docker Compose assume que usaremos imagens que estão no Docker Hub. Vamos criar uma imagem para executar programas em PHP, criar uma conta no Docker Hub e mandá-la para lá. Posteriormente, usaremos esta imagem e a imagem padrão de MySQL fornecida pelo Docker para executar este blog.

Crie uma conta no Docker Registry (https://registry.hub.docker.com/), com suas credenciais.

No repositório deste capítulo criei um diretório chamado `php5-fpm-nginx` dentro do repositório da máquina virtual para Docker. Lá

existe um `Dockerfile` que cria uma imagem de *nginx* e PHP5-FPM. Esta imagem recebe um volume externo (https://docs.docker.com/userguide/dockervolumes/) com a aplicação em PHP, portanto pode ser utilizada para servir outras aplicações além do WordPress.

Vamos usar a máquina virtual criada anteriormente. Conecte com `vagrant ssh` e vá ao diretório `/vagrant` da máquina virtual. Entre no diretório `php5-fpm-nginx` e faça o build de sua imagem:

```
$ sudo docker build -t php5-fpm-nginx .
```

Após algumas mensagens de log, a imagem estará criada localmente:

```
$ sudo docker build -t php5-fpm-nginx .
Sending build context to Docker daemon 26.17 MB
Sending build context to Docker daemon
Step 0 : FROM ubuntu:14.04

Step 1 : RUN apt-get update && apt-get -y install php5-fpm
php-config php5-mysql nginx
 ---> Using cache

Step 2 : ADD default /etc/nginx/sites-available/default
 ---> Using cache

Step 3 : CMD /etc/init.d/nginx start && php5-fpm -F
 ---> Using cache
```

Para testar, ainda dentro do mesmo diretório execute:

```
root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# docker ps
CONTAINER ID   IMAGE        COMMAND      CREATED        STATUS
PORTS          NAMES

root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# docker
run -d -p 80:80 -v /vagrant/php5-fpm/simple:/app php5-fpm-nginx


root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# docker ps
CONTAINER ID        IMAGE                   COMMAND
```

```
PORTS                NAMES
5                    php5-fpm-nginx:latest   "/bin/sh -c '/etc/in

CREATED              STATUS
2 seconds ago        Up 1 seconds        0.   0.0.0:80->80/tcp
                     elated_kirch
```

Abra o http://192.168.33.2/ no browser e veja a página de `phpinfo` gerada por esta aplicação. Vamos submeter esta imagem ao repositório público do Docker. O primeiro passo é fazer seu login, com os dados da conta que foi criada anteriormente:

```
# docker login
Username: gleicon
Password:
Email: gleicon@gmail.com
WARNING: login credentials saved in /home/vagrant/.dockercfg.
Login Succeeded
```

Existem algumas maneiras de criar uma imagem no Docker Hub. A que vamos usar é construir partir da execução um container existente. Outra maneira que veremos é o processo de build, mas quero demonstrar algumas interações com o *docker daemon*.

```
root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# sudo
docker run -d -p 80:80 php5-fpm-nginx

root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# docker ps
CONTAINER ID        IMAGE                   COMMAND


CREATED         STATUS
2 seconds ago   Up 2 seconds   0.0.0.0:80->80/tcp   serene_sammet

root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# sudo
docker commit -m "created /app" -a "Gleicon Moraes"
          1c gleicon/php5-fpm-nginx
```

O ID 1c é o ID do container que vamos usar para criar nossa imagem no Docker Hub. O próximo passo é adicionar este container a uma imagem:

```
$ sudo docker commit -m "created /app" -a "Gleicon Moraes"
```

Podemos ver as imagens criadas localmente e a imagem que queremos enviar para o Docker Hub:

```
root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# docker
images
REPOSITORY               TAG                 IMAGE ID




CREATED             VIRTUAL SIZE
8 seconds ago       256 MB
5 minutes ago       256 MB
49 minutes ago      256 MB
53 minutes ago      256 MB
57 minutes ago      256 MB
2 days ago          188.3 MB
```

Vamos testar localmente antes de fazer o `push`:

```
root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# sudo
docker run -d -p 80:80 -v /vagrant/php5-fpm/simple:/app -t -i
gleicon/php5-fpm-nginx



Bind for 0.0.0.0:80 failed: port is already allocated
root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# docker ps
CONTAINER ID        IMAGE                   COMMAND
PORTS               NAMES


CREATED              STATUS
About a minute ago   Up About a minute   0.0.0.0:80->80/tcp
                     serene_sammet

root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# docker
kill serene_sammet serene_sammet
root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# sudo
docker run -d -p 80:80 -v /vagrant/php5-fpm/simple:/app -t -i
gleicon/php5-fpm-nginx
```

```
root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx# docker ps
CONTAINER ID        IMAGE
PORTS                NAMES


COMMAND                 CREATED
"/bin/sh -c '/etc/in   2 seconds ago

STATUS
Up 1 seconds        0.0.0.0:80->80/tcp   serene_archimedes

root@vagrant-ubuntu-trusty-64:/vagrant/php5-fpm-nginx#
```

Quando tentamos executar a nossa imagem, ocorreu um erro porque a porta estava em uso. Este é um dos problemas de ter portas fixas. Com o comando `kill` matamos o container que estava em execução e tentamos novamente. Na execução, passamos o *volume* da aplicação simples. Se acessar http://192.168.33.2/ pelo browser, a página deve abrir normalmente.

Com a imagem criada, vamos enviar a nossa conta no Docker Hub. Substitua o meu login pelo seu:

```
$ sudo docker push gleicon/php5-fpm-nginx
```

Após algum tempo, sua imagem estará presente em seu repositório de forma semelhante ao meu:

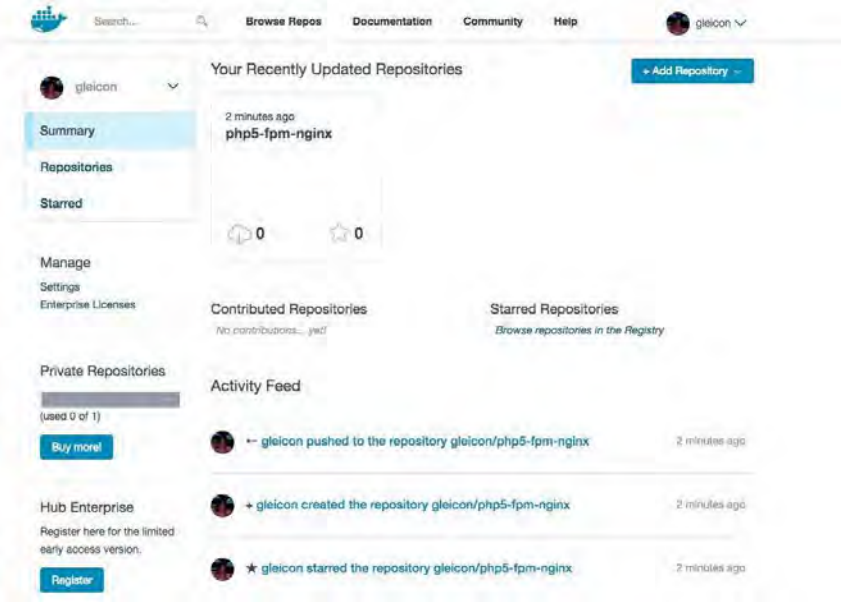


Figura 9.2: Docker Hub

A saída do comando `push` deve ser semelhante a esta:

```
$ sudo docker push gleicon/php5-fpm-nginx
The push refers to a repository [gleicon/php5-fpm-nginx] (len: 1)
```

Se quiséssemos criar a imagem no processo de build, usaríamos:

```
$ docker build -t gleicon/php5-fpm-nginx .
$ docker push gleicon/php5-fpm-nginx
```

9.6 DOCKER COMPOSE

Na mesma máquina virtual, execute a instalação do Docker Compose de acordo com as instruções do site para seu sistema operacional (https://docs.docker.com/compose/install/).

```
vagrant@vagrant-ubuntu-trusty-64:~$ sudo -s
root@vagrant-ubuntu-trusty-64:~# curl -L https://github.com/
docker/compose/releases/download/1.2.0/docker-compose-`uname
-s`-`uname -m` > /usr/local/bin/docker-compose

  % Total    % Received % Xferd  Average Speed   Time    Time
                                 Dload  Upload   Total   Spent



  Time    Current
  Left    Speed

--:--:--       k

root@vagrant-ubuntu-trusty-64:~# chmod +x
/usr/local/bin/docker-compose
root@vagrant-ubuntu-trusty-64:~# docker-compose help
Get help on a command.

Usage: help COMMAND
root@vagrant-ubuntu-trusty-64:~#
```

Só use este modo de instalação se confiar absolutamente na origem e em um ambiente isolado. Executar um `shell script` diretamente da internet não é uma boa prática.

Como exercício, você deve criar um playbook do Ansible que faça download do script, confira o md5 e faça uma instalação de forma controlada com privilégios menores do que root.

Para efeito didático, estou instalando em uma máquina virtual isolada do meu ambiente de desenvolvimento e de produção para evitar problemas de comprometimento.

Em seguida, vamos fazer download da versão mais recente do WordPress:

```
$ mkdir wordpress-compose
```

```
$ cd wordpress-compose/
$ curl https://wordpress.org/latest.tar.gz | tar -xvzf -
$ ls -l
ls -l
total 4
drwxr-xr-x 5 nobody nogroup 4096 Apr 27 17:14 wordpress
```

Vamos usar nosso container de PHP e o container de MySQL padrão do Docker. Temos que criar a configuração do compose e fazer uma pequena alteração no WordPress. Para tanto, vamos criar neste diretório um arquivo chamado `docker-compose.yml` :

```
db:
  image: mysql:latest
  expose:
    - "   6"
  ports:
    - "   7:   6"
  environment:
    MYSQL_DATABASE: wordpress
    MYSQL_USER: wordpress
    MYSQL_PASSWORD: wordpress
    MYSQL_ROOT_PASSWORD: aaa123
web:
  image: gleicon/php5-fpm-nginx
  ports:
    - "80:80"
  links:
    - db
  volumes:
    - ./wordpress:/app
```

Este arquivo usa o mesmo formato que o Ansible, `YAML` , e descreve dois componentes da nossa arquitetura: `web` e `db` . O componente `web` é a nossa imagem de `php5-fpm-nginx` com o WordPress. O MySQL é uma imagem oficial mantida pelo time do Docker. Existem imagens oficiais para muitos programas, vale a pena explorar o Docker Hub. Junto aos dados da imagem do MySQL passei os dados de usuários e senha.

Na configuração do componente `web` , o item chamado *links* avisa que este container depende do container `db` . Não é uma

dependência de ordem de execução mas de ligação.

Os dados dos componentes da arquitetura serão disponibilizados dentro do ambiente do container, em variáveis de ambiente. O mesmo tipo que criamos com o comando `export` do bash. As aplicações terão IPs inválidos e estes dados só estarão disponíveis no momento de execução. É uma maneira de não ter que gerar arquivos dinâmicos de configuração ou trocá-los a todo momento.

Vamos editar um arquivo do WordPress para adequá-los a este paradigma. Em uma instalação de produção, este passo deve ser automatizado e versionado para possibilitar upgrades e *patches* de segurança.

Dentro do diretório `wordpress`, crie o arquivo `wp-config.php` com o conteúdo a seguir:

```
<?php
// **
  MySQL settings - You can get this info from your web host
** //
/** The name of the database for WordPress */
define('DB_NAME', 'wordpress');

/** MySQL database username */
define('DB_USER', 'wordpress');

/** MySQL database password */
define('DB_PASSWORD', 'wordpress');

/** MySQL hostname */
define('DB_HOST', 'db:   6');

/** Database Charset to use in creating database tables. */
define('DB_CHARSET', 'utf8');

/** The Database Collate type. Don't change this if in doubt. */
define('DB_COLLATE', '');

/* MySQL database table prefix. */
$table_prefix = 'wp_';
```

```
/* Authentication Unique Keys and Salts. */
/* https://api.wordpress.org/secret-key/1.1/salt/ */
define( 'AUTH_KEY',         'put your unique phrase here' );
define( 'SECURE_AUTH_KEY',  'put your unique phrase here' );
define( 'LOGGED_IN_KEY',    'put your unique phrase here' );
define( 'NONCE_KEY',        'put your unique phrase here' );
define( 'AUTH_SALT',        'put your unique phrase here' );
define( 'SECURE_AUTH_SALT', 'put your unique phrase here' );
define( 'LOGGED_IN_SALT',   'put your unique phrase here' );
define( 'NONCE_SALT',       'put your unique phrase here' );


/* WordPress Localized Language. */
define( 'WPLANG', '' );


/* Absolute path to the WordPress directory. */
if ( !defined('ABSPATH') )
    define('ABSPATH', dirname(__FILE__) . '/');

/* Sets up WordPress vars and included files. */
require_once(ABSPATH . 'wp-settings.php');
```

A variável `DB_HOST` tem o valor `db`, que será apresentado ao container `web` como o IP do container de MySQL. O `docker-compose` cria entradas para evitar o uso do IP direto na configuração.

Vamos executar o `docker-compose`:

```
$ docker-compose up
```

Após alguns minutos, você deve ter uma saída parecida com esta, seguida de logs de aplicação:

```
root@vagrant-ubuntu-trusty-64:/vagrant/wordpress-compose#
docker-compose up
Creating wordpresscompose_db_1...
Pulling image mysql:latest...
latest: Pulling from mysql
```

```
mysql:latest: The image you are pulling has been verified.
Important: image verification is a tech preview feature and
should not be relied on to provide security.


Status: Downloaded newer image for mysql:latest
Creating wordpresscompose_web_1...
Attaching to wordpresscompose_db_1, wordpresscompose_web_1
```

No seu browser, acesse http://192.168.33.2:800/ e a instalação básica do WordPress deve aparecer. O seu terminal ficou com os logs dos dois containers pois não executamos o `docker-compose` com `detach`. Vamos aproveitar esta visão para navegar no WordPress e ver os logs produzidos. Para interromper os containers, use `CONTROL+C`.

Após a primeira execução, você pode reiniciar com o comando:

```
$ sudo docker-compose start
Starting wordpresscompose_db_1...
Starting wordpresscompose_web_1...
```

Em outro terminal conectado à máquina virtual execute `vagrant ssh` e os comandos a seguir:

```
docker ps
CONTAINER ID        IMAGE
PORTS               NAMES



COMMAND                  CREATED
"/bin/sh -c '/etc/in   3 minutes ago
"/entrypoint.sh mysq   3 minutes ago
```

```
STATUS
Up About a minute   0.0.0.0:80->80/tcp    wordpresscompose_web_1
```

Com os nomes dos containers, vamos investigar as variáveis de ambiente em cada um deles. Elas foram criadas pelo `docker-compose`. Para executar comandos dentro de um container em execução, usamos `docker exec <nome-do-container> env`.

```
root@vagrant-ubuntu-trusty-64:/vagrant/wordpress-compose#
docker exec wordpresscompose_web_1 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

```
WORDPRESSCOMPOSE_DB_1_PORT_3306_TCP_PROTO=tcp
WORDPRESSCOMPOSE_DB_1_NAME=
  /wordpresscompose_web_1/wordpresscompose_db_1
WORDPRESSCOMPOSE_DB_1_ENV_MYSQL_PASSWORD=wordpress
WORDPRESSCOMPOSE_DB_1_ENV_MYSQL_ROOT_PASSWORD=aaa123
WORDPRESSCOMPOSE_DB_1_ENV_MYSQL_USER=wordpress
```

```
WORDPRESSCOMPOSE_DB_1_ENV_MYSQL_DATABASE=wordpress
WORDPRESSCOMPOSE_DB_1_ENV_MYSQL_MAJOR=5.6
WORDPRESSCOMPOSE_DB_1_ENV_MYSQL_VERSION=5.6.24
HOME=/root
```

```
root@vagrant-ubuntu-trusty-64:/vagrant/wordpress-compose#
  docker exec wordpresscompose_db_1 env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

MYSQL_PASSWORD=wordpress
MYSQL_ROOT_PASSWORD=aaa123
MYSQL_USER=wordpress
MYSQL_DATABASE=wordpress
MYSQL_MAJOR=5.6
MYSQL_VERSION=5.6.2
HOME=/root
```

Usando as variáveis de ambiente exportadas no container, poderíamos criar uma aplicação com zero configuração por arquivo. Todos os dados dos bancos e do ambiente estão lá.

No container do banco, temos as variáveis que declaramos no `docker-compose.yml`, `MYSQL_DATABASE`. Vamos inspecionar os processos do container `web`:

```
root@vagrant-ubuntu-trusty-64:/vagrant/wordpress-compose#
  docker exec wordpresscompose_web_1 ps -ef
UID        PID  PPID  C STIME TTY
```

```
TIME CMD
00:00:00 /bin/sh -c /etc/init.d/nginx start && php5-fpm -F
00:00:00 nginx: master process /usr/sbin/nginx
00:00:00 nginx: worker process
00:00:00 nginx: worker process
00:00:00 nginx: worker process
```

```
00:00:00 nginx: worker process
00:00:00 php-fpm: master process (/etc/php5/fpm/php-fpm.conf)
00:00:00 php-fpm: pool www
00:00:00 php-fpm: pool www
00:00:00 ps -ef
```

Podemos nos conectar da maquina virtual ao MySQL do container:

```
$ sudo apt-get install mysql-client
root@vagrant-ubuntu-trusty-64:/vagrant/wordpress-compose#
mysql --protocol=TCP -u wordpress --host=localhost -P 3
wordpress -p
Enter password:
Reading table information for completion of table and column
names
You can turn off this feature to get a quicker startup with -A

Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.6.24 MySQL Community Server (GPL)

Copyright (c) 2000, 2015, Oracle and/or its affiliates. All
rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current
input statement.

mysql> show tables;
+-----------------------+
| Tables_in_wordpress   |
+-----------------------+
| wp_commentmeta        |
| wp_comments           |
| wp_links              |
| wp_options            |
| wp_postmeta           |
| wp_posts              |
| wp_term_relationships |
| wp_term_taxonomy      |
| wp_terms              |
| wp_usermeta           |
| wp_users              |
+-----------------------+
```

```
11 rows in set (0.00 sec)

mysql> select post_title from wp_posts;
+--------------+
| post_title   |
+--------------+
| Hello world! |
| Sample Page  |
| Auto Draft   |
| el posto     |
| el posto     |
+--------------+
5 rows in set (0.00 sec)
```

Para finalizar, vamos usar o que aprendemos no capítulo anterior para provocar uma carga no nosso blog e examinar como os processos aparecem na máquina virtual que está rodando o Docker. Em um terminal local fora da máquina virtual, execute:

```
$ while true; do ab -c 20 -n 200 http://192.168.33.2 /; sleep 5;
done
```

Observe que os processos de PHP5-FPM e MySQL aparecem no top do host, mas quando executamos `ps -ef` no container `web` não vimos processos do host lá.

```
top - 21:16:18 up  4:00,  1 user,  load average: 2.14, 0.62, 0.27
Tasks: 114 total,   7 running, 107 sleeping,   0 stopped,   0 zombie
%Cpu(s): 64.3 us, 31.7 sy,  0.0 ni,  0.0 id,  0.0 wa,  4.0 hi,  0.0 si,  0.0 st
KiB Mem:   1017868 total,   939784 used,    78084 free,    14556 buffers
KiB Swap:        0 total,        0 used,        0 free.   237672 cached Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
24706 www-data  20   0  231508  20336  13232 R 17.9  2.0   0:02.67 php5-fpm
24708 www-data  20   0  231508  20336  13232 R 17.9  2.0   0:02.44 php5-fpm
24709 www-data  20   0  231252  20080  13232 R 17.9  2.0   0:00.58 php5-fpm
24704 www-data  20   0  231256  20240  13232 R 17.6  2.0   0:04.33 php5-fpm
24710 www-data  20   0  230996  19992  13228 R 12.3  2.0   0:00.37 php5-fpm
24402 999       20   0  795960 463960   7688 S  9.6 45.6   0:05.29 mysqld
24494 www-data  20   0   86380   2504    936 S  2.7  0.2   0:00.79 nginx
    3 root      20   0       0      0      0 S  1.0  0.0   0:04.68 ksoftirqd/0
    8 root      20   0       0      0      0 S  1.0  0.0   0:07.21 rcuos/0
    7 root      20   0       0      0      0 R  0.7  0.0   0:07.61 rcu_sched
24354 root      20   0       0      0      0 S  0.3  0.0   0:01.03 kworker/0:1
24705 root      20   0   23656   1588   1104 R  0.3  0.2   0:00.05 top
    1 root      20   0   33636   2912   1388 S  0.0  0.3   0:04.54 init
    2 root      20   0       0      0      0 S  0.0  0.0   0:00.02 kthreadd
    5 root       0 -20       0      0      0 S  0.0  0.0   0:00.00 kworker/0:0H
    9 root      20   0       0      0      0 S  0.0  0.0   0:00.00 rcu_bh
   10 root      20   0       0      0      0 S  0.0  0.0   0:00.01 rcuob/0
   11 root      rt   0       0      0      0 S  0.0  0.0   0:00.01 migration/0
   12 root      rt   0       0      0      0 S  0.0  0.0   0:00.53 watchdog/0
```

Figura 9.3: top no host do Docker

Se executarmos o comando `ps` novamente:

```
$ sudo docker exec wordpresscompose_web_1 ps -ef
```

O buffer de log do kernel terá a seguinte mensagem:

```
$ sudo dmesg -T
....

apparmor="DENIED" operation="ptrace" profile="docker-default"
pid=25136 comm="ps" requested_mask="trace" denied_mask="trace"
peer="docker-default"
```

Este é um dos dispositivos para controlar o acesso de aplicações e containers sendo executados em isolamento de namespaces.

9.7 CONCLUSÃO

Após o Docker, outras ferramentas de virtualização e isolamento de processos surgiram e grandes empresas como Amazon, Microsoft e Google investiram tempo e dinheiro para prover serviços e ferramentas compatíveis.

O Ansible possui uma interface básica com o Docker e em alguns provedores de serviço você pode executar suas imagens diretamente, sem ter que comprar uma máquina virtual.

Containers também são a base de serviços como o Heroku, uma plataforma para aplicações web. É uma ferramenta importante e não tão nova que está passando por um renascimento.

Vale a pena entender como os containers podem ajudar na flexibilidade e eficiência de sua aplicação. Neste tópico, conhecimentos de sistema operacional, performance, profiling e arquitetura se misturam e não podem viver sem automação sob o risco de procedimentos complexos impedirem o crescimento e operação dos seus sistemas. Lembre-se de que a coleta de métricas também facilita o uso e a criação de containers sob demanda.

CAPÍTULO 10

EM PRODUÇÃO

Este capítulo tem algumas considerações sobre o uso do que aprendemos em produção. Tudo o que vimos durante o livro foi pensado e criado com base em experiência real de produção, mas cada organização tem suas regras. O objetivo das ferramentas que vimos é aumentar a eficiência de forma inteligente.

10.1 DEPLOY

Vamos começar pelo que o Ansible chama de *inventory file*, o nosso `hosts.ini`. Você pode organizar suas máquinas de maneiras diferentes, mas eu recomendo que o inventory file reflita a arquitetura do seu sistema com nomes significativos, por exemplo:

```
[loadbalancer]
10.0.0.1
[web]
10.0.0.2
10.0.0.3
[db]
10.0.0.4
10.0.0.5
```

Nos capítulos em que fizeram deploy na AWS e DigitalOcean, vimos *dynamic inventories* nos quais o `hosts.ini` era substituído por scripts ou dados em memória. O Ansible possui uma API que pode ser acessada com a linguagem Python e estes dados podem ser armazenados em *datastores* como Redis e MySQL.

Recomendo também estudar o Ansible Tower e outros

componentes do ecossistema para gerenciar seus serviços. Você pode construir APIs que gerem os inventory files dinamicamente e coletar dados de uso das APIs de IaaS disponíveis em seu provedor. Vale a pena aprender a usar a linguagem Python para construir módulos do Ansible.

Alguns provedores como *Digital Ocean*, *Azure* e *Amazon* oferecem códigos de descontos e camadas de serviço com pouco ou nenhum custo. Aproveite estas ofertas para testar os playbooks e criar seus próprios playbooks em um ambiente de produção. Lembre de criar a monitoração dos seus sistemas.

10.2 BUILDING BLOCKS E LOCK-IN

Building Blocks são unidades que podem ser compostas para construir componentes mais complexos. É importante reduzir o uso de building blocks a elementos comuns como máquinas virtuais, *load balancers*, volumes de storage e imagens. Pensando assim, não é difícil encontrar equivalência entre os provedores de serviço, que geralmente dão nomes e medidas distintas ao seus produtos.

Para demonstrar e cobrir um provedor de serviços novo, vamos criar um conjunto de duas máquinas virtuais e um load balancer no Azure. Com a camada grátis que oferecem, é possível experimentar seus produtos, mas vamos nos limitar ao mais simples e sem automação, para entender a interface do provedor:

Após criar sua conta, entre no painel e selecione *Virtual Machines*. Crie duas máquinas com o perfil mais simples, sistema operacional Ubuntu 14.04 LTS. Pode usar o atalho com um grande símbolo de + no canto inferior esquerdo:

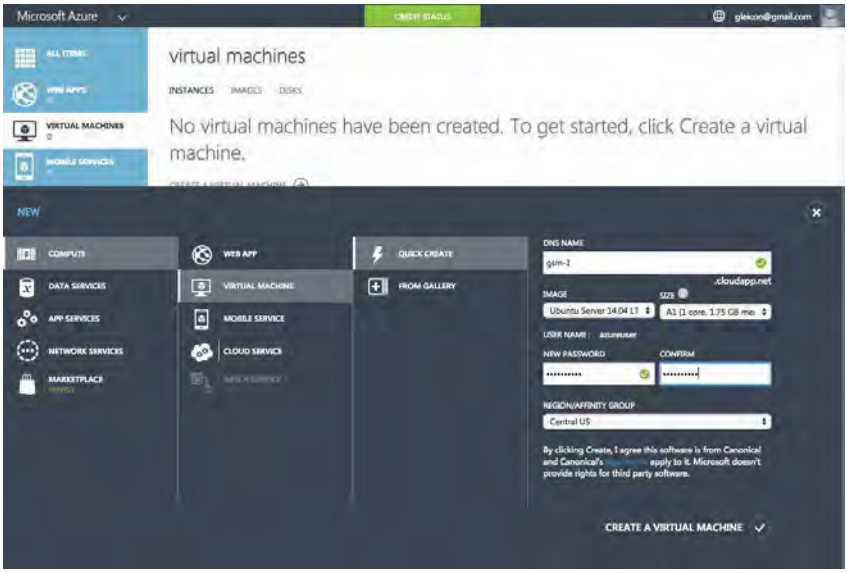


Figura 10.1: Azure Virtual Machines

Use prefixos que façam sentido para você, no meu caso `gsm-1` e `gsm-2`. Vamos criar um load balancer, que no Azure é chamado de *Traffic Manager* (*Elastic LoadBalancer* ou ELB, no AWS):

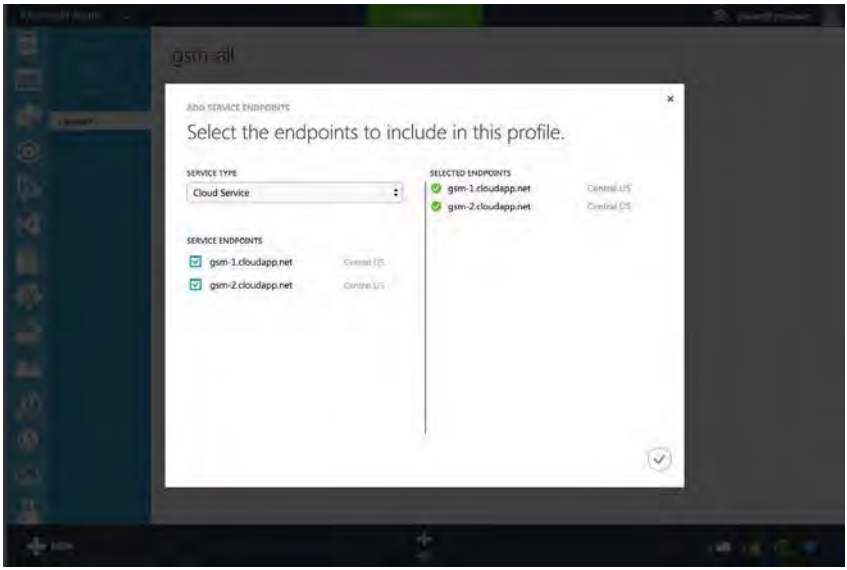


Figura 10.2: Azure Traffic Manager

Escolhi o prefixo `gsm-all`. O Traffic Manager fica dentro de *Network Services*. O nosso fica em `gsm-all.trafficmanager.net`. No dashboard do Traffic Manager, podemos ir em *endpoints* e ver que o status é *degraded*, o load balancer não consegue se conectar.

Precisamos instalar o *nginx* nas máquinas e abrir a porta 80 no firewall de cada uma.

```
$ ssh azureuser@gsm-1.cloudapp.net
$ sudo apt-get update
$ sudo apt-get install nginx
$ sudo -s
# echo "gsm-1" > /usr/share/nginx/html/index.html

$ ssh azureuser@gsm-2.cloudapp.net
$ sudo apt-get update
$ sudo apt-get install nginx
$ sudo -s
# echo "gsm-2" > /usr/share/nginx/html/index.html
```

Na interface do Azure vá até Virtual Machines e selecione cada uma das máquinas. Na sessão *endpoints*, clique em `+ ADD` no rodapé

da página. Preencha a regra como a figura a seguir:

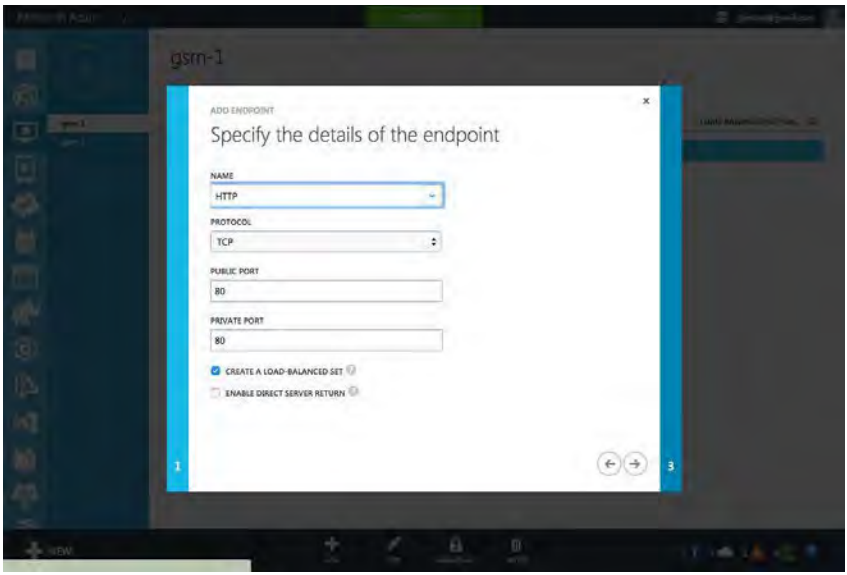


Figura 10.3: Azure FW Rule Creation

Repita na segunda máquina. Após a criação das regras, o Traffic Manager deve ter saído do estado *degraded*. Vamos executar um teste local:

```
$ while true; do curl gsm-all.trafficmanager.net; sleep 5; done
```

A saída deste teste deve alternar algumas vezes com `gsm-1` e `gsm-2` (ou as strings que você colocou). Não é dificil automatizar esta configuração com Ansible ou qualquer outro sistema mas meu objetivo foi navegar por um provedor de serviço diferente do que usamos anteriormente e identificar os mesmos *building blocks*.

Você poderia ter sua aplicação em ambos e mover carga de produção caso tivesse problemas ou o preço estivesse melhor. Isso ajuda a se prevenir ou pelo menos aliviar os efeitos do *Lock-in*.

Lock-in é a medida que diz qual a dependência de sua aplicação

e negócio ao provedor de serviços. Uma das variáveis é o volume de dados que você tem em um provedor, que pode ser complexo de replicar ou transferir para fora dele. Pior, pode ser caro pois alguns serviços não cobram pela ingestão e trafego local de dados mas cobram pela saída de dados ou pela banda consumida. Outra variável importante é o quanto de código você teria que fazer para substituir um serviço utilizado como por exemplo filas ou caches.

Um diagrama de blocos simples pode ajudar a visualizar estes pontos e planejar o pior caso possível: building blocks simples como máquinas virtuais com capacidade limitada e load balancer. É bom desenvolver um plano com estas limitações para estimar se é possível migrar para uma estrutura interna ou para outro provedor. Muitas empresas investem em criar seu **cloud privado** por preocupações com segurança ou confidencialidade e é difícil chegar perto do trabalho que provedores como Azure, Google e AWS oferecem.

10.3 BLUE/GREEN DEPLOY E PACOTES

Ensaie um processo simples de deploy imaginando que sua estrutura está funcionando e tem clientes em produção. Sugiro começar pelo *blue/green* deploy que nada mais é que duplicar sua estrutura e utilizar o loadbalancer como ponto de troca de tráfego. Leia mais neste post do Martin Fowler (http://martinfowler.com/bliki/BlueGreenDeployment.html).

Esta técnica deve ser adaptada à aplicação e ao ambiente. Em alguns casos, gosto de manter o mesmo banco de dados para as duas instâncias do sistema se as mudanças forem incrementais, em outros é melhor pesquisar uma maneira de sincronizar dados para ter os ambientes completamente separados.

Em alguns casos, precisamos empacotar as aplicações para o

formato nativo do sistema operacional. Isso pode ser interessante se, no seu pipeline de deploy, suas aplicações precisam do ok de um time como QA ou Segurança. Pesquise sobre o empacotamento de aplicações e avalie se não vale a pena criar arquivos `.deb` ou `rpm` de suas aplicações juntamente com um repositório local.

Pode ser que para alguns componentes, por exemplo o *nginx*, você precise de uma versão customizada ou queira trocar por equivalentes como o OpenResty (http://openresty.org/). Com o Ansible, você pode automatizar a criação destes pacotes. O FPM (https://github.com/jordansissel/fpm) é uma aplicação bem legal para gerar pacotes com pouco atrito. Pode ser um passo intermediário se você optou por criar imagens para cada versão de sua aplicação.

Em provedores como a Amazon, você pode ter uma biblioteca de imagens personalizadas. Pode ser mais simples ter uma sequência versionada de imagens da sua aplicação para aproveitar recursos como *auto-scaling* em vez de reconstruir o ambiente toda vez que precisar de mais capacidade. A velocidade de alocação de capacidade e a complexidade da construção de cada elemento do ambiente deve ser levada em conta.

10.4 TESTES

Em seu pipeline pode existir um ponto em que você usa um servidor de integração como Jenkins (https://jenkins-ci.org/) ou Travis-CI (https://travis-ci.org) verificar testes unitários e de integração. Se você ainda não usa este serviço, pesquise e comece a usar. Existem modalidades SaaS e instaladas em suas máquinas e elas podem contribuir com mais do que executar testes: podem ser um ponto inicial para implementar automações de entrega e deploy. Você sempre terá três ambientes: **desenvolvimento**, **homologação** e **produção**. Eles podem ser o mesmo ambiente cumprindo estas três

funções, o que não é recomendável, ou segregados.

Procure testes e métricas que possam ser aplicados em seu ambiente de produção para que o desenvolvimento reflita as condições reais de produção. Sugiro duas fontes: *Chaos Monkeys released into the Wild* (http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html) e *Metrics Everywhere* (https://www.youtube.com/watch?v=czes-oa0yik). Chaos monkeys foi a resposta da Netflix à ideia de introduzir mudanças em ambientes de produção para descobrir conexões frágeis entre os elementos de sua arquitetura; métricas é a resposta de desenvolvedores a outro tipo de monitoração que não seja reativa, por exemplo contar tempo de resposta a requests HTTP em vez de apenas monitorar a porta 80.

Benchmarks normalmente são inconclusivos se o ambiente e a metodologia aplicada não demonstram as diferenças entre cenários ou componentes (comparar dois frameworks HTTP não diz muito além de que eles são dois frameworks diferentes). Por outro lado, vale a pena executar testes simples de carga e performance em seus ambientes de produção e até mesmo antes e depois de uma mudança significativa de código em seu ambiente de homologação. Uma ferramenta interessante é o *AB* (Apache Benchmark). Ele está instalado por default no Mac OS X e tem pacotes para a maioria das distribuições Linux.

A figura a seguir mostra uma execução que fiz direcionando ao https://www.google.com/ com os parâmetros `-n 100 -c 10` .

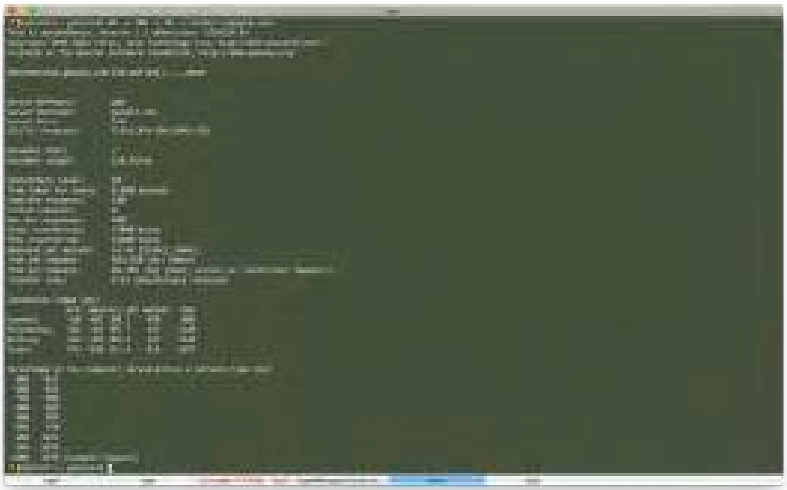


Figura 10.4: Apache Bench no Google

O Apache Benchmark representa de forma sumarizada dados que serão encontrados em outros sistemas de teste de carga e benchmark. Eu informei que queria 100 requests (`-n 100`) com 10 clientes simultâneos (`-c 10`). O primeiro bloco informa versões de software, tipos de códigos de retorno, requests por segundo, tempo médio (mediana) e transferência. Logo abaixo, uma quebra do tempo de conexões entre conexão TCP/IP, processamento e espera.

O último bloco é chamado de *Percentage of the requests served within a certain time (ms)* e tem a informação mais importante da execução, que são os percentuais e tempos organizados em faixas. A coluna da esquerda tem um percentual, e a da esquerda, um tempo. Significa que a quantidade especificada (50%, 60%, 70% etc.) das conexões teve tempo abaixo do valor à direita. Esta informação é melhor que a média de tempo de conexões pois nos altos percentuais (90%, 95%, 99% e 99.9%) podem se esconder tempos de resposta que causam insatisfação e a impressão de má qualidade para o usuário de sua aplicação.

Execute o AB contra sua aplicação utilizando diferentes tamanhos de instâncias virtuais e também localmente com o Vagrant para avaliar o impacto do poder de processamento e I/O de sua infraestrutura no resultado final de sua aplicação.

Como vimos anteriormente, usando o New Relic para monitorar as respostas e o AB para gerar carga sintética conseguimos entender como o conjunto de aplicação e banco de dados funciona.

Existem vários sistemas de teste de carga e tempo de resposta. Eu recomento conhecer o `locust.io` (http://locust.io/). É um sistema feito em Python que pode ser executado local ou remotamente.

Eu preparei um projeto para facilitar seu uso na AWS (https://github.com/gleicon/locust-swarm) que orquestra e cria um sistema com máquinas virtuais para executar os testes do *locust.io*. Os agentes de teste são programas em Python que podem ser elaborados e simular logins, buscas e atividades que o usuário faria em sua aplicação.

A figura a seguir é a tela de entrada do Locust, onde informamos o número de usuários que devem ser simulados e a taxa de criação (*hatching*) deles:

Figura 10.5: Novo teste no Locust

Um teste em execução:

Figura 10.6: Novo teste no Locust

Exploramos estas ferramentas anteriormente e o desafio é pensar em como elas poderiam ser integrados na sua rotina de build

e testes. Como ensaiar um perfil de tráfego projetado para sua aplicação e interrompê-la em pontos que são opacos ao código ? Vimos exemplos de como reduzir a banda de rede de uma máquina virtual e injetar uma carga sintética. Como poderíamos fazer isso com a CPU e I/O?

10.5 CONCLUSÃO

Este capítulo é um apanhado de conceitos que vimos durante o livro e alguns temas que não consegui abordar, mas achei que deveriam ser pelo menos citados.

Tentei mostrar neste livro um pouco mais do que as ferramentas e introduzir um framework inicial de trabalho para automação de sistemas. Administração de sistemas é uma área divertida e tem muito em comum com desenvolvimento de sistemas.

O tempo de reação a problemas em produção e com clientes é uma das diferenças e pode criar situações tensas mas a aplicação de automação e de novas ideias neste campo tem tornado o trabalho bem mais eficiente e seguro. É uma oportunidade única de ver o impacto de bugs e otimizações em produção.

As últimas gerações de sistemas têm ficado complexas, se não pelo seu objetivo, pelo menos pelo fato de que são sistemas que precisam de mais de uma máquina para serem executados.

Não tem como abordar a tarefa de construir e manter estes ambientes de forma manual e com paradigmas que usávamos há dez ou vinte anos. Por outro lado, as ferramentas vão além de scripts enigmáticos e combinação de scripts *perl* com ferramentas de sistemas.

Boa sorte!