Capítulo 4

INFRAESTRUTURA COMO CÓDIGO

Administradores de sistema aprendem o poder da linha de comando desde cedo e passam boa parte do dia no terminal controlando a infraestrutura da sua empresa. É comum dizer que eles estão alterando a **configuração do sistema**, seja instalando novos servidores, instalando novo software, fazendo *upgrade* de software existente, editando arquivos de configuração, reiniciando processos, lendo arquivos de log, efetuando e restaurando *backups*, criando novas contas de usuário, gerenciando permissões, ajudando desenvolvedores a investigar algum problema ou escrevendo *scripts* para automatizar tarefas repetíveis.

Cada uma dessas tarefas altera ligeiramente o estado do servidor e pode potencialmente causar problemas se a mudança não for executada corretamente. É por isso que muitos administradores de sistema não dão acesso aos servidores de produção para desenvolvedores: para evitar mudanças indesejáveis e pela falta de confiança. No entanto, em muitas empresas, essas tarefas são executadas manualmente nos servidores de produção pelos próprios administradores de sistema e eles também podem cometer erros.

Um dos principais princípios de DevOps é investir em automação. Automação permite executar tarefas mais rapidamente e diminui a possibilidade de erros humanos. Um processo automatizado é mais confiável e pode ser auditorado com mais facilidade.

Conforme o número de servidores que precisam ser administrados aumenta, processos automatizados se tornam essenciais. Alguns administradores de sistema escrevem seus próprios *scripts* para automatizar as tarefas mais comuns. O problema é que cada um escreve sua própria versão dos *scripts*, e fica difícil reutilizá-los em outras situações. Por outro lado, desenvolvedores são bons em escrever código modularizado e reutilizável.

Com o avanço da cultura DevOps e o aumento da colaboração entre administradores de sistema e desenvolvedores, diversas ferramentas têm evoluído para tentar padronizar o gerenciamento automatizado de infraestrutura. Tais ferramentas permitem tratar infraestrutura da mesma forma que tratamos código: usando controle de versões, realizando testes, empacotando e distribuindo módulos comuns e, obviamente, executando as mudanças de configuração no servidor.

Essa prática é conhecida como **infraestrutura como código** e nos permitirá resolver o problema que encontramos no final do capítulo anterior. Em vez de reinstalar tudo manualmente, vamos usar uma ferramenta de gerenciamento de configurações para automatizar o processo de provisionamento, configuração e *deploy* da loja virtual.

4.1 PROVISIONAMENTO, CONFIGURAÇÃO OU DEPLOY?

Ao comprar um laptop ou um celular novo, você só precisa ligá-lo e ele já está pronto para uso. Em vez de mandar as peças individuais e um manual de como montar sua máquina, o fabricante já fez todo o trabalho necessário para que você usufrua seu novo aparelho sem maiores problemas. Como usuário, você só precisa instalar programas e restaurar seus arquivos para considerar a máquina sua. Esse processo de preparação para o usuário final é conhecido como **provisionamento**.

O termo **provisionamento** é comumente usado por empresas de telecomunicações e por equipes de operações para se referir às etapas de preparação iniciais de configuração de um novo recurso: provisionar um aparelho celular, provisionar acesso à internet, provisionar um servidor, provisionar uma nova conta de usuário, e assim por diante. Provisionar um aparelho celular envolve, entre outras coisas: alocar uma nova linha, configurar os equipamentos de rede que permitem completar ligações, configurar serviços extra como SMS ou e-mail e, por fim, associar tudo isso com o chip que está no seu celular.

No caso de servidores, o processo de provisionamento varia de empresa para empresa dependendo da infraestrutura e da divisão de responsabilidades entre as equipes. Se a empresa possui infraestrutura própria, o processo de provisionamento envolve a compra e a instalação física do novo servidor no seu *data center*. Se a empresa possui uma infraestrutura virtualizada, o processo de provisionamento só precisa alocar uma nova máquina virtual para o servidor.

Da mesma forma, se a equipe de operações considerar a equipe de desenvolvimento como "usuário final", o processo de provisionamento acaba quando o servidor está acessível na rede, mesmo se a aplicação em si ainda não estiver rodando. Se considerarmos os verdadeiros usuários como "usuários finais", então o processo de provisionamento só acaba quando o servidor e a aplicação estiverem rodando e acessíveis na rede.

Para evitar maior confusão e manter integridade no decorrer do livro, vale a pena olhar para o processo de ponta a ponta e definir uma terminologia adequada. Imaginando o cenário mais longo de uma empresa que possui infraestrutura própria, mas que não tem servidores sobrando para uso, as etapas necessárias para colocar uma nova aplicação no ar seriam:

1. **Compra de hardware**: em empresas grandes, essa etapa inicia um processo de aquisição, que envolve diversas justificações e aprovações, pois investir dinheiro em hardware influi na contabilidade e no planejamento financeiro da empresa.
2. **Instalação física do hardware**: essa etapa envolve montar o novo servidor em um *rack* no *data center*, assim como instalação de cabos de força, de rede etc.
3. **Instalação e configuração do sistema operacional**: uma vez que o servidor é ligado, é preciso instalar um sistema operacional e configurar os itens básicos de hardware como: interfaces de rede, armazenamento (disco, partições, volumes em rede), autenticação e autorização de usuários, senha de *root*, repositório de pacotes etc.
4. **Instalação e configuração de serviços comuns**: além das configurações base do sistema operacional, muitos

servidores precisam configurar serviços de infraestrutura básicos como: DNS, NTP, SSH, coleta e rotação de logs, backups, firewall, impressão etc.

5. **Instalação e configuração da aplicação**: por fim, é preciso instalar e configurar tudo o que fará este servidor ser diferente dos outros: componentes de middleware, a aplicação em si, assim como configurações do middleware e da aplicação.

No decorrer do livro, usaremos o termo **provisionamento** para se referir ao processo que envolve as etapas 1 a 4, ou seja, todas as atividades necessárias para que o servidor possa ser usado e independente da razão pela qual ele foi requisitado. Usaremos o termo **deploy**, ou o equivalente em português **implantação**, quando nos referirmos à etapa 5.

Apesar do *deploy* precisar de um servidor provisionado, o ciclo de vida do servidor é diferente da aplicação. O mesmo servidor pode ser usado por várias aplicações e cada aplicação pode efetuar *deploy* dezenas ou até centenas de vezes, enquanto o provisionamento de novos servidores acontece com menos frequência.

No pior caso, o processo de provisionamento pode demorar semanas ou até meses, dependendo da empresa. Por esse motivo, é comum ver equipes de desenvolvimento definindo com bastante antecedência os requisitos de hardware para seus ambientes de produção e iniciando esse processo com a equipe de operações nas etapas iniciais do projeto, para evitar atrasos quando a aplicação estiver pronta para ir ao ar. Alguma colaboração pode acontecer no começo do processo, porém a maior parte do tempo as duas

equipes trabalham em paralelo, sem saber o que a outra está fazendo.

Pior ainda, muitas dessas decisões arquiteturais cruciais sobre a quantidade de servidores, configurações de hardware ou escolha do sistema operacional são tomadas no começo do projeto, no momento em que se tem a menor quantidade de informação e conhecimento sobre as reais necessidades do sistema.

Um dos aspectos mais importantes da cultura DevOps é reconhecer esse conflito de objetivos e criar um ambiente de colaboração entre as equipes de desenvolvimento e de operações. O compartilhamento de práticas e ferramentas permite que a arquitetura se adapte e evolua conforme as equipes aprendem mais sobre o sistema rodando no mundo real: em produção. Esta visão holística também ajuda a encontrar soluções que simplifiquem o processo de compra e provisionamento, removendo etapas ou utilizando automação para fazer com que elas aconteçam mais rápido.

Um bom exemplo dessa simplificação do processo é o uso de tecnologias como virtualização de hardware e computação em nuvem — também conhecida pelo termo em inglês *cloud computing* ou simplesmente *cloud*. Provedores de serviço *cloud* de ponta permitem que você tenha um novo servidor no ar em questão de minutos através de um simples clique de botão ou chamada de API!

4.2 FERRAMENTAS DE GERENCIAMENTO DE CONFIGURAÇÃO

Na seção anterior, explicamos a diferença entre provisionamento e *deploy*. Com exceção dos itens 1 e 2 que envolvem compra e instalação física de hardware, as outras três etapas envolvem algum tipo de configuração do sistema: seja do sistema operacional em si, do software base instalado na máquina ou da aplicação que roda em cima dele. A figura a seguir mostra as diferentes etapas do ciclo de vida de um servidor e o escopo do gerenciamento de configuração.

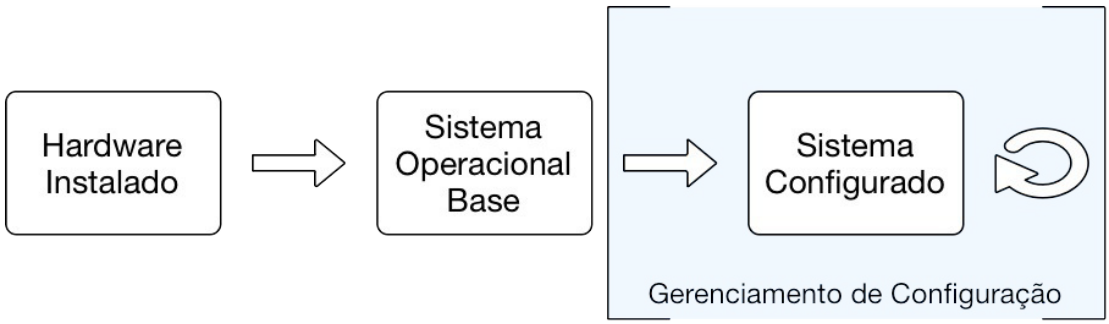

Figura 4.1: Escopo do gerenciamento de configuração

Existem diversas ferramentas para gerenciamento de configuração. Algumas das mais populares são: ***shell scripts*** (a linha de comando também é conhecida como *shell*), **Puppet** (http://www.puppetlabs.com/puppet/), **Chef** (http://www.opscode.com/chef/), **Ansible** (http://www.ansibleworks.com/tech/), **Salt** (http://saltstack.com/community), **Pallet** (http://palletops.com/) e **CFEngine** (http://cfengine.com/).

A maioria dessas ferramentas possui uma empresa responsável pelo seu desenvolvimento e pelo crescimento da comunidade e ecossistema ao seu redor. Todas elas estão contribuindo para a evolução da comunidade DevOps e, apesar de competirem entre si, seu maior adversário ainda é a falta de uso de ferramentas. Por

incrível que pareça, a maioria das empresas ainda gerencia seus servidores de forma manual, ou possui *scripts* proprietários escritos há tanto tempo que ninguém consegue dar manutenção.

A grande diferença entre *shell scripts* proprietários e as outras ferramentas é que geralmente o *script* só funciona para instalar e configurar o servidor pela primeira vez. Ele serve como documentação executável dos passos executados na primeira instalação. Mas se o *script* for executado novamente, ele provavelmente não vai funcionar. O administrador de sistemas precisa ser bem disciplinado e precisa escrever código extra para lidar com as situações nas quais algum pacote já está instalado, ou precisa ser removido ou atualizado.

As outras ferramentas especializadas têm uma característica importante, conhecida como **idempotência**: você pode executá-las diversas vezes seguidas com o mesmo código e elas mudarão apenas o que for necessário. Se um pacote já está instalado, ele não será reinstalado. Se um serviço já está rodando, ele não será reiniciado. Se o arquivo de configuração já possui o conteúdo correto, ele não será alterado.

Idempotência permite escrever código de infraestrutura de forma declarativa. Em vez da instrução ser *"instale o pacote X"* ou *"crie o usuário Y"*, você diz *"eu quero que o pacote X esteja instalado"* ou *"eu quero que o usuário Y exista"*. Você declara o estado final desejado e, quando a ferramenta executa, caso o pacote ou o usuário já existam, nada acontecerá.

A próxima coisa que você pode estar se perguntando é: *"qual dessas ferramentas devo usar?"*. Neste capítulo, usaremos o Puppet por permitir exemplificar bem essa propriedade declarativa da

linguagem, por ser uma ferramenta madura, aceita na comunidade de administradores de sistema e por ser bem adotada na comunidade.

Porém, a verdade é que usar qualquer uma dessas ferramentas é melhor do que nada. As ferramentas nesse espaço estão evoluindo constantemente e é importante você se familiarizar com algumas delas antes de tomar a decisão de qual é melhor para sua situação.

4.3 INTRODUÇÃO AO PUPPET: RECURSOS, PROVEDORES, MANIFESTOS E DEPENDÊNCIAS

Cada uma das ferramentas de gerenciamento de configuração possui uma terminologia própria para se referir aos elementos da linguagem e aos componentes do seu ecossistema. Em um nível fundamental, cada comando que você declara na linguagem é uma **diretiva** e você pode definir um conjunto delas em um **arquivo de diretivas**, que é o equivalente a um arquivo de código-fonte de uma linguagem de programação como Java ou Ruby.

No Puppet, essas diretivas são chamadas de **recursos**. Alguns exemplos de recursos que você pode declarar no Puppet são: pacotes, arquivos, usuários, grupos, serviços, trechos de *script* executável, dentre outros. O arquivo no qual você declara um conjunto de diretivas é chamado de **manifesto**. Ao escrever código Puppet, você passará a maior parte do tempo criando e editando esses manifestos.

Para se familiarizar com a sintaxe da linguagem Puppet e seu

modelo de execução, vamos reconstruir a máquina virtual do servidor de banco de dados e logar por SSH na máquina que está zerada:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
==> db: Importing base box 'hashicorp/precise32'...
==> db: Matching MAC address for NAT networking...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: Setting the name of the VM: blank_db_1394854529922_29194
==> db: Clearing any previously set network interfaces...
==> db: Preparing network interfaces based on configuration...
    db: Adapter 1: nat
    db: Adapter 2: hostonly
==> db: Forwarding ports...
    db: 22 => 2222 (adapter 1)
==> db: Running 'pre-boot' VM customizations...
==> db: Booting VM...
==> db: Waiting for machine to boot. This may take a few
        minutes...
    db: SSH address: 127.0.0.1:2222
    db: SSH username: vagrant
    db: SSH auth method: private key
==> db: Machine booted and ready!
==> db: Configuring and enabling network interfaces...
==> db: Mounting shared folders...
    db: /vagrant => /private/tmp/blank
$ vagrant ssh db
Welcome to Ubuntu 12.04 LTS (GNU/Linux ...
vagrant@db$
```

Aproveitando que a *box* do Vagrant já possui o Puppet instalado, podemos começar a usá-lo diretamente criando um novo arquivo de manifesto vazio chamado `db.pp`. Por padrão, manifestos possuem extensão `.pp`:

```
vagrant@db$ nano db.pp
```

Como conteúdo inicial, vamos declarar um recurso para o pacote `mysql-server` e dizer que o queremos instalado no

sistema. Isso é feito passando o valor `installed` para o parâmetro `ensure` do recurso `package`:

```
package { "mysql-server":
  ensure => installed,
}
```

Se você comparar essa declaração de recurso com o comando que executamos no capítulo *Tudo começa em produção* (`sudo apt-get install mysql-server`), verá que eles são parecidos. A principal diferença é a natureza **declarativa** da linguagem do Puppet. No comando manual, instruímos o gerenciador de pacotes a instalar o pacote `mysql-server` , enquanto que no Puppet declaramos o estado final desejado do sistema.

Manifestos não são uma lista de comandos a ser executados. Quando o Puppet executa, ele compara o estado atual do sistema com o estado declarado no manifesto, calcula a diferença e executa apenas as mudanças necessárias. Após salvar o arquivo, execute o Puppet com o seguinte comando:

```
vagrant@db$ sudo puppet apply db.pp
err: /Stage[main]//Package[mysql-server]/ensure: change from
purged to present failed: Execution of ...
...
E: Unable to fetch some archives, maybe run apt-get update or ...

notice: Finished catalog run in 6.62 seconds
```

Você pode ver que o Puppet tentou alterar o estado do pacote `mysql-server` de removido (`purged`) para presente (`present`), no entanto, encontrou erros pois esquecemos de executar o comando `apt-get update` , como fizemos no capítulo *Tudo começa em produção*. Para isso, vamos declarar um novo recurso `exec` e definir uma dependência dizendo que ele deve

rodar antes do pacote `mysql-server`. Mudaremos o conteúdo do arquivo `db.pp` para:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
  require => Exec["apt-update"],
}
```

Para entender o conteúdo do manifesto, precisamos conhecer mais algumas características da linguagem do Puppet. Todo recurso possui um nome — a string entre aspas duplas logo após a declaração e antes do `:` (dois pontos) — que serve como um identificador único. Quando um recurso precisa se referir a outro, usamos a sintaxe com a primeira letra maiúscula e o nome do recurso entre colchetes.

O Puppet não garante que a ordem de execução respeite a ordem em que os recursos são declarados no manifesto. Quando existe uma dependência, você precisa declará-la explicitamente. Neste caso, adicionamos o parâmetro `require` no recurso `package` para garantir que o comando `apt-get update` execute antes da instalação do MySQL. Agora o Puppet instalará o pacote `mysql-server` quando executado novamente:

```
vagrant@db$ sudo puppet apply db.pp
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//Package[mysql-server]/ensure: ensure \
        changed 'purged' to 'present'
notice: Finished catalog run in 59.98 seconds
```

Você pode ver que o Puppet alterou o estado do pacote `mysql-server` de removido (`purged`) para presente

(`present`). Para confirmar que o pacote foi realmente instalado, você pode executar o comando:

```
vagrant@db$ aptitude show mysql-server
Package: mysql-server
State: installed
...
```

Perceba que o gerenciador de pacotes diz que o estado atual do pacote `mysql-server` é "instalado" (*installed*). Para entender a natureza declarativa do manifesto, tente rodar o Puppet novamente:

```
vagrant@db$ sudo puppet apply db.pp
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: Finished catalog run in 6.41 seconds
```

Dessa vez, o recurso `apt-update` executou, porém nada aconteceu com o pacote `mysql-server` , que já estava instalado, satisfazendo o estado declarado em nosso manifesto do Puppet.

Outra diferença que você pode ter percebido entre o manifesto Puppet e o comando manual do capítulo *Tudo começa em produção* é que não precisamos dizer para o Puppet qual gerenciador de pacotes usar. Isso acontece pois recursos do Puppet são abstratos.

Existem várias implementações diferentes (chamadas de **provedores**) para os diversos sistemas operacionais. Para o recurso `package` , o Puppet possui diversos provedores: `apt` , `rpm` , `yum` , `gem` , dentre outros.

Na hora da execução, com base em informações do sistema, o Puppet vai escolher o provedor mais adequado. Para ver uma lista com todos os provedores disponíveis para um determinado

recurso, assim como a documentação detalhada dos parâmetros que o recurso aceita, você pode executar o comando:

```
vagrant@db$ puppet describe package

package
=======
Manage packages. ...
...
Providers
---------
    aix, appdmg, apple, apt, aptitude, aptrpm, blastwave,
    dpkg, fink, freebsd, gem, hpux, macports, msi, nim,
    openbsd, pacman, pip, pkg, pkgdmg, pkgutil, portage,
    ports, portupgrade, rpm, rug, sun, sunfreeware, up2date,
    urpmi, yum, zypper
```

Agora que entendemos o funcionamento básico do Puppet, vamos parar de executar comandos manuais dentro do servidor e começar a escrever código de infraestrutura fora da máquina virtual. Usaremos o Vagrant para nos ajudar a provisionar o servidor.

Recomeçar usando Puppet com Vagrant

Primeiramente vamos sair da máquina virtual `db` e destruí-la novamente, como fizemos no fim do capítulo *Monitoramento*:

```
vagrant@db$ logout
Connection to 127.0.0.1 closed.
$ vagrant destroy db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Forcing shutdown of VM...
==> db: Destroying VM and associated drives...
```

No mesmo diretório em que você criou o arquivo `Vagrantfile`, crie um novo chamado `manifests` e, dentro dele, um novo arquivo `db.pp` com o mesmo conteúdo da seção

anterior. A nova estrutura de diretórios deve estar assim:

```
.
├── Vagrantfile
└── manifests
    └── db.pp
```

Agora é preciso alterar o arquivo de configuração do Vagrant, o `Vagrantfile`, para usar o Puppet como ferramenta de provisionamento da máquina virtual `db`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.hostname = "db"
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    web_config.vm.hostname = "web"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.hostname = "monitor"
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

Feito isso, assim que subirmos uma nova máquina virtual para o servidor de banco de dados, o Vagrant rodará o Puppet automaticamente, sem precisarmos logar no servidor por SSH e

executar comandos manualmente. A saída da execução do Puppet será mostrada logo após a saída normal do Vagrant e você perceberá se algum erro acontecer. Por exemplo, a execução inicial mostrará o pacote `mysql-server` sendo instalado:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
==> db: Importing base box 'hashicorp/precise32'...
==> db: Matching MAC address for NAT networking...
==> db: Checking if box 'hashicorp/precise32' is up to date...
==> db: Setting the name of the VM: blank_db_1394854957677_21160
==> db: Clearing any previously set network interfaces...
==> db: Preparing network interfaces based on configuration...
    db: Adapter 1: nat
    db: Adapter 2: hostonly
==> db: Forwarding ports...
    db: 22 => 2222 (adapter 1)
==> db: Running 'pre-boot' VM customizations...
==> db: Booting VM...
==> db: Waiting for machine to boot. This may take a few \
        minutes...
    db: SSH address: 127.0.0.1:2222
    db: SSH username: vagrant
    db: SSH auth method: private key
==> db: Machine booted and ready!
==> db: Configuring and enabling network interfaces...
==> db: Mounting shared folders...
    db: /vagrant => /private/tmp/blank
    db: /tmp/vagrant-puppet-2/manifests => \
        /private/tmp/blank/manifests
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//Package[mysql-server]/ensure: ensure \
        changed 'purged' to 'present'
notice: Finished catalog run in 71.68 seconds
```

Agora que sabemos como usar o Vagrant e o Puppet para provisionar nosso servidor, é hora de restaurar o ambiente de produção da loja virtual.

4.4 REINSTALANDO O SERVIDOR DE BANCO DE DADOS

Temos um manifesto simples que executa o primeiro passo da instalação do servidor de banco de dados. Se você lembra do próximo passo que tomamos no capítulo *Tudo começa em produção*, precisamos nesse momento criar o arquivo de configuração `/etc/mysql/conf.d/allow_external.cnf` para permitir acesso remoto ao servidor MySQL.

Faremos isso usando o recurso `file`, que representa a existência de um arquivo no sistema. Alteramos o conteúdo do arquivo `db.pp` para:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
  require => Exec["apt-update"],
}

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => "[mysqld]\n  bind-address = 0.0.0.0",
  require => Package["mysql-server"],
}
```

Por padrão, o nome do recurso representa o caminho completo onde o arquivo será criado no sistema, e o parâmetro `content` possui o conteúdo do arquivo. Para especificar um caminho diferente, você pode usar o parâmetro `path`. Os parâmetros `owner`, `group` e `mode` representam o usuário e o grupo associados ao arquivo e suas permissões.

Para aplicar a nova configuração, em vez de destruir e subir uma nova máquina virtual, você pode usar o comando `provision` do Vagrant para simplesmente executar o Puppet na máquina virtual já existente:

```
$ vagrant provision db
  ==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d \
                            /allow_external.cnf]/ \
        ensure: defined content as \
        '{md5}4149205484cca052a3bfddc8ae60a71e'
notice: Finished catalog run in 4.36 seconds
```

Dessa vez, o Puppet criou o arquivo `/etc/mysql/conf.d/allow_external.cnf` . Para saber quando um arquivo de configuração precisa ser alterado, ele calcula um *checksum* MD5 do seu conteúdo. O MD5 é um algoritmo de *hash* bastante usado para verificar a integridade de um arquivo. Uma pequena alteração no conteúdo do arquivo faz com que o seu *checksum* MD5 seja totalmente diferente.

No nosso caso, o conteúdo do arquivo de configuração é pequeno e conseguimos declará-lo por completo no manifesto. No entanto, é comum ter arquivos de configuração com dezenas ou centenas de linhas. Gerenciar arquivos longos como uma única string em código Puppet não é uma boa ideia. Podemos trocar o conteúdo do arquivo por um **template**. O Puppet entende *templates* ERB (*embedded ruby*), um sistema de *template* que permite embutir código Ruby dentro de um arquivo texto.

Para usar um *template*, primeiramente precisamos extrair o

conteúdo para um novo arquivo `allow_ext.cnf`, que deve ficar junto ao manifesto:

```
.
├── Vagrantfile
└── manifests
    ├── allow_ext.cnf
    └── db.pp
```

Mude o conteúdo do arquivo `allow_ext.cnf` ligeiramente, para que o Puppet altere o arquivo na próxima execução:

```
[mysqld]
   bind-address = 9.9.9.9
```

Por fim, trocamos o parâmetro `content` do recurso `file` para usar o *template*:

```
exec ...
package ...

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
}
```

Ao executar o Puppet novamente, você verá que o *checksum* MD5 mudará e o conteúdo do arquivo será alterado:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d \
                            /allow_external.cnf]/ \
        content: content changed \
                 '{md5}4149205484cca052a3bfddc8ae60a71e' to \
```

```
                '{md5}8e2381895fcf40fa7692e38ab86c7192'
notice: Finished catalog run in 4.60 seconds
```

Agora precisamos reiniciar o serviço para que o MySQL perceba a alteração no arquivo de configuração. Fazemos isso declarando um novo recurso do tipo `service`:

```
exec ...
package ...
file ...

service { "mysql":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["mysql-server"],
}
```

O serviço possui novos parâmetros: `ensure => running`, que garante que o serviço esteja rodando; `enable`, que garante que o serviço rode sempre que o servidor reiniciar; `hasstatus` e `hasrestart`, que declaram que o serviço entende os comandos `status` e `restart`; por fim, o parâmetro `require`, que declara uma dependência com o recurso `Package["mysql-server"]`.

Se você tentar executar o Puppet, verá que nada acontece, pois o serviço já foi iniciado quando o pacote foi instalado. Para reiniciar o serviço toda vez que o arquivo de configuração mudar, precisamos declarar uma nova dependência no recurso `File["/etc/mysql/conf.d/allow_external.cnf"]`:

```
exec ...
package ...

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
```

```
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
  notify  => Service["mysql"],
}

service ...
```

O parâmetro `notify` define uma dependência de execução: sempre que o recurso `file` for alterado, ele fará com que o recurso `service` execute. Dessa vez, se alterarmos o conteúdo do *template* `allow_ext.cnf` de volta para `0.0.0.0` e executarmos o Puppet, o serviço será reiniciado:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /etc/mysql/conf.d \
                            /allow_external.cnf]/ \
        content: content changed \
                 '{md5}8e2381895fcf40fa7692e38ab86c7192' to \
                 '{md5}907c91cba5e1c7770fd430182e42c437'
notice: /Stage[main]//Service[mysql]: Triggered 'refresh' \
        from 1 events
notice: Finished catalog run in 6.88 seconds
```

Com essa alteração, o banco de dados está rodando novamente e uma das verificações do Nagios deve ficar verde. Para corrigir a próxima verificação, precisamos completar a configuração do banco de dados criando o *schema* e o usuário da loja virtual.

Criaremos o *schema* declarando mais um recurso `exec` com o mesmo comando usado no capítulo *Tudo começa em produção*:

```
exec ...
package ...
file ...
service ...
```

```
exec { "loja-schema":
  unless  => "mysql -uroot loja_schema",
  command => "mysqladmin -uroot create loja_schema",
  path    => "/usr/bin/",
  require => Service["mysql"],
}
```

Além da dependência com o recurso `Service["mysql"]`, usamos um novo parâmetro `unless` que especifica um comando de teste: caso seu código de saída seja zero, o comando principal não vai executar. Essa é a forma de tornar um recurso do tipo `exec` idempotente. Ao contrário do `exec` que usamos para rodar o `apt-get update`, neste caso é importante que o Puppet não tente criar um novo *schema* toda vez que for executado.

Conforme fizemos no capítulo *Tudo começa em produção*, precisamos remover a conta anônima de acesso ao MySQL antes de criar o usuário para acessar nosso novo *schema*. Faremos isso declarando um novo recurso `exec`:

```
exec ...
package ...
file ...
service ...
exec ...

exec { "remove-anonymous-user":
  command => "mysql -uroot -e \"DELETE FROM mysql.user \
                                WHERE user=''; \
                                FLUSH PRIVILEGES\"",
  onlyif  => "mysql -u' '",
  path    => "/usr/bin",
  require => Service["mysql"],
}
```

Neste comando, em vez de usar o parâmetro `unless`, usamos o parâmetro oposto `onlyif`, que vai executar o comando principal apenas se o código de saída do comando de teste for zero.

Este comando de teste tenta acessar o MySQL usando um usuário vazio. Caso consiga conectar, o Puppet precisa executar o comando SQL `DELETE ...` para remover a conta anônima. Por fim, criaremos um último recurso `exec` para criar o usuário com permissão de acesso ao *schema* da loja virtual:

```
exec ...
package ...
file ...
service ...
exec ...
exec ...

exec { "loja-user":
  unless  => "mysql -uloja -plojasecret loja_schema",
  command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                                loja_schema.* TO 'loja'@'%' \
                                IDENTIFIED BY 'lojasecret';\"",
  path    => "/usr/bin/",
  require => Exec["loja-schema"],
}
```

Ao executar o Puppet pela última vez no servidor de banco de dados, veremos uma saída parecida com:

```
$ vagrant provision db
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//Exec[loja-schema]/returns: executed
        successfully
notice: /Stage[main]//Exec[loja-user]/returns: executed
        successfully
notice: /Stage[main]//Exec[remove-anonymous-user]/returns:
        executed \ successfully
notice: Finished catalog run in 15.56 seconds
```

Com isso, conseguimos restaurar por completo o servidor de banco de dados e todas as verificações do Nagios referentes ao *host*

`db` deverão ficar verdes. A única verificação que permanecerá vermelha é a do Tomcat, pois a aplicação ainda não está acessível.

Nas próximas seções, automatizaremos o processo de provisionamento do servidor `web` e do *deploy* da aplicação. Por enquanto, o conteúdo completo do manifesto `db.pp` ficou:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { "mysql-server":
  ensure  => installed,
  require => Exec["apt-update"],
}

file { "/etc/mysql/conf.d/allow_external.cnf":
  owner   => mysql,
  group   => mysql,
  mode    => 0644,
  content => template("/vagrant/manifests/allow_ext.cnf"),
  require => Package["mysql-server"],
  notify  => Service["mysql"],
}

service { "mysql":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["mysql-server"],
}

exec { "loja-schema":
  unless  => "mysql -uroot loja_schema",
  command => "mysqladmin -uroot create loja_schema",
  path    => "/usr/bin/",
  require => Service["mysql"],
}

exec { "remove-anonymous-user":
  command => "mysql -uroot -e \"DELETE FROM mysql.user \
```

```
                                WHERE user=''; \
                                FLUSH PRIVILEGES\"",
  onlyif  => "mysql -u' '",
  path    => "/usr/bin",
  require => Service["mysql"],
}

exec { "loja-user":
  unless  => "mysql -uloja -plojasecret loja_schema",
  command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                                loja_schema.* TO 'loja'@'%' \
                                IDENTIFIED BY 'lojasecret';\"",
  path    => "/usr/bin/",
  require => Exec["loja-schema"],
}
```

4.5 REINSTALANDO O SERVIDOR WEB

Agora que o servidor `db` está restaurado, vamos automatizar o processo de provisionamento e *deploy* no servidor `web`. Para isto, criaremos um novo arquivo no diretório `manifests` chamado `web.pp`:

```
.
├── Vagrantfile
└── manifests
    ├── allow_ext.cnf
    ├── db.pp
    └── web.pp
```

Em vez de associar o novo manifesto com o servidor `web` já existente, vamos criar uma nova máquina virtual temporária chamada `web2` para podermos testar nosso código Puppet isoladamente antes de aplicá-lo no ambiente de produção. Ao final do capítulo, nos livraremos da máquina virtual `web2` e aplicaremos a mesma configuração no servidor `web` de verdade.

O conteúdo do `Vagrantfile` com a nova máquina virtual

deve ficar:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.hostname = "db"
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    web_config.vm.hostname = "web"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
  end

  config.vm.define :web2 do |web_config|
    web_config.vm.hostname = "web2"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.13"
    web_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "web.pp"
    end
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.hostname = "monitor"
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

Inicialmente vamos declarar apenas os recursos para executar o `apt-get update` e instalar os pacotes base do servidor `web2`, de forma similar aos comandos executados no capítulo *Tudo começa em produção*:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { ["mysql-client", "tomcat7"]:
  ensure  => installed,
  require => Exec["apt-update"],
}
```

Ao subir a nova máquina virtual, o Puppet vai aplicar essa configuração e instalar os pacotes:

```
$ vagrant up web2
Bringing machine 'web2' up with 'virtualbox' provider...
==> web2: Importing base box 'hashicorp/precise32'...
...
==> web2: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed
        successfully
notice: /Stage[main]//Package[mysql-client]/ensure: ensure
        changed \ 'purged' to 'present'
notice: /Stage[main]//Package[tomcat7]/ensure: ensure changed \
        'purged' to 'present'
notice: Finished catalog run in 105.89 seconds
```

Feito isso, você pode abrir seu navegador, digitar a URL `http://192.168.33.13:8080/` e você deverá ver uma página que diz *"It Works!"*. Isso significa que o Tomcat está instalado e rodando corretamente.

Os próximos passos realizados no capítulo *Tudo começa em produção* foram a configuração da conexão segura com HTTPS, a criação do `keystore` para armazenar o certificado SSL e o aumento da quantidade de memória disponível para a máquina virtual Java quando o Tomcat inicia.

Todas essas operações envolvem editar ou criar um novo

arquivo no sistema. Já sabemos como entregar arquivos com o Puppet, então faremos os três passos de uma só vez. Primeiramente, vamos reaproveitar os arquivos de configuração que já estão funcionando no servidor `web` como *templates* para o manifesto do Puppet. Os seguintes comandos vão copiar os arquivos da máquina virtual `web` para o diretório compartilhado pelo Vagrant:

```
$ vagrant ssh web -- 'sudo cp /var/lib/tomcat7/conf/.keystore \
> /vagrant/manifests/'
$ vagrant ssh web -- 'sudo cp /var/lib/tomcat7/conf/server.xml \
> /vagrant/manifests/'
$ vagrant ssh web -- 'sudo cp /etc/default/tomcat7 \
> /vagrant/manifests/'
```

Em vez de usar o comando `vagrant ssh` para logar na máquina virtual, passamos um comando entre aspas após os traços `--`. Esse comando executará dentro da máquina virtual. O caminho `/vagrant` é onde o Vagrant mapeia o diretório no qual você definiu o `Vagrantfile` dentro da máquina virtual. Se tudo der certo, os arquivos fora do Vagrant devem ser:

```
.
├── Vagrantfile
└── manifests
    ├── .keystore
    ├── allow_ext.cnf
    ├── db.pp
    ├── server.xml
    ├── tomcat7
    └── web.pp
```

Como não precisamos alterar nada no conteúdo desses arquivos, basta declararmos novos recursos do tipo `file` para que o Puppet entregue-os no lugar certo, ajustando seus donos e permissões:

```
exec ...
package ...

file { "/var/lib/tomcat7/conf/.keystore":
  owner   => root,
  group   => tomcat7,
  mode    => 0640,
  source  => "/vagrant/manifests/.keystore",
  require => Package["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/server.xml",
  require => Package["tomcat7"],
}

file { "/etc/default/tomcat7":
  owner   => root,
  group   => root,
  mode    => 0644,
  source  => "/vagrant/manifests/tomcat7",
  require => Package["tomcat7"],
}
```

Por último, precisamos declarar o serviço `tomcat7`, que precisa ser reiniciado quando os arquivos de configuração são alterados. Os parâmetros do recurso `service` são os mesmos que usamos no serviço `mysql` quando configuramos o servidor de banco de dados:

```
exec ...
package ...

file { "/var/lib/tomcat7/conf/.keystore":
  ...
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
```

```
  ...
  notify  => Service["tomcat7"],
}

file { "/etc/default/tomcat7":
  ...
  notify  => Service["tomcat7"],
}

service { "tomcat7":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["tomcat7"],
}
```

Podemos então rodar o manifesto todo para aplicar as últimas configurações no servidor `web2` :

```
$ vagrant provision web2
==> web2: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /server.xml]/ content: content \
        changed '{md5}523967040584a921450af2265902568d' to \
                '{md5}e4f0d5610575a720ad19fae4875e9f2f'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /.keystore]/ ensure: defined \
        content as '{md5}80792b4dc1164d67e1eb859c170b73d6'
notice: /Stage[main]//File[ /etc/default \
                            /tomcat7]/ content: content \
        changed '{md5}49f3fe5de425aca649e2b69f4495abd2' to \
        '{md5}1f429f1c7bdccd3461aa86330b133f51'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' \
        from 3 events
notice: Finished catalog run in 14.58 seconds
```

Feito isso, temos o Tomcat configurado para aceitar conexões HTTP e HTTPS. Podemos testar abrindo uma nova janela no

navegador web e acessando a URL `https://192.168.33.13:8443/`. Após aceitar o certificado autoassinado, você verá uma página que diz *"It works!"*.

4.6 FAZENDO DEPLOY DA APLICAÇÃO

Para termos um servidor `web` completo, precisamos fazer o *deploy* da loja virtual. Em vez de usar o Puppet para reconstruir o processo de *build* dentro da máquina virtual — como fizemos no capítulo *Tudo começa em produção* —, vamos reaproveitar o artefato `.war` gerado. No capítulo *Integração contínua*, revisitaremos essa decisão e encontraremos uma forma melhor de gerar o artefato `.war` da aplicação.

Vamos também reaproveitar o *template* do arquivo `context.xml`, no qual definimos os recursos JNDI para acesso ao banco de dados. Usando o mesmo truque da seção anterior, vamos copiar os arquivos de dentro da máquina virtual `web`:

```
$ vagrant ssh web -- 'sudo cp \
> /var/lib/tomcat7/conf/context.xml /vagrant/manifests/'
$ vagrant ssh web -- 'sudo cp \
> /var/lib/tomcat7/webapps/devopsnapratica.war \
> /vagrant/manifests/'
```

Agora o conteúdo dos arquivos e diretórios fora do Vagrant devem ser:

```
.
├── Vagrantfile
└── manifests
    ├── .keystore
    ├── allow_ext.cnf
    ├── context.xml
    ├── db.pp
    ├── devopsnapratica.war
```

```
    ├── server.xml
    ├── tomcat7
    └── web.pp
```

O artefato contento a aplicação web `devopsnapratica.war` não precisa ser alterado, porém vamos definir variáveis dentro do manifesto para conter os dados de acesso ao banco de dados. Essas variáveis serão substituídas quando o Puppet processar o *template* ERB que criaremos no arquivo `context.xml`:

```
exec ...
package ...
file ...
file ...
file ...
service ...

$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"

file { "/var/lib/tomcat7/conf/context.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  content => template("/vagrant/manifests/context.xml"),
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
  owner   => tomcat7,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/devopsnapratica.war",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}
```

A sintaxe do Puppet para declarar variáveis é um cifrão `$` seguido do nome da variável. Usamos o símbolo de igual `=` para

atribuir valores às nossas variáveis. Para usar essas variáveis dentro do *template* do arquivo `context.xml`, precisamos alterar o seu conteúdo substituindo os valores existentes pelo marcador `<%= var %>`, em que `var` deve ser o nome da variável sem o cifrão `$`. O novo *template* ERB do arquivo `manifests/context.xml` deve ser:

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>

    <Resource name="jdbc/web" auth="Container"
      type="javax.sql.DataSource" maxActive="100"
      maxIdle="30"
      maxWait="10000"
      username="<%= db_user %>"
      password="<%= db_password %>"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>

    <Resource name="jdbc/secure" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000"
      username="<%= db_user %>"
      password="<%= db_password %>"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>

    <Resource name="jdbc/storage" auth="Container"
      type="javax.sql.DataSource" maxActive="100" maxIdle="30"
      maxWait="10000"
      username="<%= db_user %>"
      password="<%= db_password %>"
      driverClassName="com.mysql.jdbc.Driver"
      url="jdbc:mysql://<%= db_host %>:3306/<%= db_schema %>"/>
</Context>
```

Com isso, finalizamos o manifesto Puppet para provisionar e fazer *deploy* da aplicação no servidor `web2`. Podemos executá-lo rodando o comando:

```
$ vagrant provision web2
==> web2: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /context.xml]/ content: content \
        changed '{md5}4861cda2bbf3a56fbfdb78622c550019' to \
                '{md5}8365f14d15ddf99b1d20f9672f0b98c7'
notice: /Stage[main]//File[ /var/lib/tomcat7/webapps \
                            /devopsnapratica.war]/ ensure: \
        defined content as \
        '{md5}cba179ec04e75ce87140274b0aaa2263'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' \
        from 2 events
notice: Finished catalog run in 21.78 seconds
```

Tente agora acessar a URL `http://192.168.33.13:8080/devopsnapratica/` no seu navegador para ver que a loja está funcionando corretamente. Sucesso!

Agora que sabemos recriar o servidor `web` por completo, podemos nos livrar da máquina virtual `web2`. Para demonstrar o poder do uso de automação para provisionamento e *deploy*, vamos também destruir a máquina virtual `web` e recriá-la do zero:

```
$ vagrant destroy web web2
    web2: Are you sure you want to destroy the 'web2' VM?[y/N] y
==> web2: Forcing shutdown of VM...
==> web2: Destroying VM and associated drives...
==> web2: Running cleanup tasks for 'puppet' provisioner...
    web: Are you sure you want to destroy the 'web' VM? [y/N] y
==> web: Forcing shutdown of VM...
==> web: Destroying VM and associated drives...
```

Antes de reiniciar a máquina virtual novamente, precisamos atualizar o arquivo de configuração do Vagrant, o `Vagrantfile`, para conter apenas um servidor `web`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  config.vm.box = "hashicorp/precise32"

  config.vm.define :db do |db_config|
    db_config.vm.hostname = "db"
    db_config.vm.network :private_network,
                         :ip => "192.168.33.10"
    db_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    web_config.vm.hostname = "web"
    web_config.vm.network :private_network,
                          :ip => "192.168.33.12"
    web_config.vm.provision "puppet" do |puppet|
      puppet.manifest_file = "web.pp"
    end
  end

  config.vm.define :monitor do |monitor_config|
    monitor_config.vm.hostname = "monitor"
    monitor_config.vm.network :private_network,
                              :ip => "192.168.33.14"
  end
end
```

O último passo é reconstruir a máquina virtual `web` do zero:

```
$ vagrant up web
Bringing machine 'web' up with 'virtualbox' provider...
==> web: Importing base box 'hashicorp/precise32'...
==> web: Matching MAC address for NAT networking...
==> web: Checking if box 'hashicorp/precise32' is up to date...
==> web: Setting the name of the VM:
         blank_web_1394856878524_4521
==> web: Fixed port collision for 22 => 2222. Now on port 2200.
==> web: Clearing any previously set network interfaces...
==> web: Preparing network interfaces based on configuration...
    web: Adapter 1: nat
    web: Adapter 2: hostonly
```

```
==> web: Forwarding ports...
    web: 22 => 2200 (adapter 1)
==> web: Running 'pre-boot' VM customizations...
==> web: Booting VM...
==> web: Waiting for machine to boot. This may take a few
         minutes...
    web: SSH address: 127.0.0.1:2200
    web: SSH username: vagrant
    web: SSH auth method: private key
==> web: Machine booted and ready!
==> web: Configuring and enabling network interfaces...
==> web: Mounting shared folders...
    web: /vagrant => /private/tmp/blank
    web: /tmp/vagrant-puppet-2/manifests =>
         /private/tmp/blank/manifests
==> web: Running provisioner: puppet...
Running Puppet with web.pp...
stdin: is not a tty
notice: /Stage[main]//Exec[apt-update]/returns: executed \
        successfully
notice: /Stage[main]//Package[mysql-client]/ensure: ensure \
        changed 'purged' to 'present'
notice: /Stage[main]//Package[tomcat7]/ensure: ensure \
        changed 'purged' to 'present'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /context.xml]/content: content \
        changed '{md5}4861cda2bbf3a56fbfdb78622c550019' to \
        '{md5}8365f14d15ddf99b1d20f9672f0b98c7'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /server.xml]/content: content \
        changed '{md5}523967040584a921450af2265902568d' to \
        '{md5}e4f0d5610575a720ad19fae4875e9f2f'
notice: /Stage[main]//File[ /var/lib/tomcat7/conf \
                            /.keystore]/ensure: defined \
        content as '{md5}80792b4dc1164d67e1eb859c170b73d6'
notice: /Stage[main]//File[ /var/lib/tomcat7/webapps \
                            /devopsnapratica.war]/ensure: \
        defined content as \
        '{md5}cba179ec04e75ce87140274b0aaa2263'
notice: /Stage[main]//File[ /etc/default \
                            /tomcat7]/content: content \
        changed '{md5}49f3fe5de425aca649e2b69f4495abd2' to \
        '{md5}1f429f1c7bdccd3461aa86330b133f51'
notice: /Stage[main]//Service[tomcat7]: Triggered 'refresh' \
        from 5 events
```

```
notice: Finished catalog run in 106.31 seconds
```

Pronto! Conseguimos reconstruir o servidor `web` e fazer *deploy* da loja virtual **em 1 minuto**!

A verificação do Nagios deve ficar verde e a loja virtual está novamente acessível para nossos usuários. A versão final do manifesto `web.pp`, ao final deste capítulo, ficou:

```
exec { "apt-update":
  command => "/usr/bin/apt-get update"
}

package { ["mysql-client", "tomcat7"]:
  ensure  => installed,
  require => Exec["apt-update"],
}

file { "/var/lib/tomcat7/conf/.keystore":
  owner   => root,
  group   => tomcat7,
  mode    => 0640,
  source  => "/vagrant/manifests/.keystore",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/conf/server.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/server.xml",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/etc/default/tomcat7":
  owner   => root,
  group   => root,
  mode    => 0644,
  source  => "/vagrant/manifests/tomcat7",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
```

```
}

service { "tomcat7":
  ensure     => running,
  enable     => true,
  hasstatus  => true,
  hasrestart => true,
  require    => Package["tomcat7"],
}

$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"

file { "/var/lib/tomcat7/conf/context.xml":
  owner   => root,
  group   => tomcat7,
  mode    => 0644,
  content => template("/vagrant/manifests/context.xml"),
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
  owner   => tomcat7,
  group   => tomcat7,
  mode    => 0644,
  source  => "/vagrant/manifests/devopsnapratica.war",
  require => Package["tomcat7"],
  notify  => Service["tomcat7"],
}
```

Investir em automação nos permitiu destruir e reconstruir servidores em questão de minutos. Você também deve ter percebido que passamos muito pouco tempo executando comandos manuais dentro dos servidores. Transferimos a responsabilidade de execução de comandos para o Puppet e declaramos o estado final desejado em arquivos de manifesto.

O código do Puppet está começando a crescer e não temos uma

estrutura clara para organizar os arquivos de manifesto e os *templates*. Também estamos tratando o artefato `.war` como um arquivo estático, algo que sabemos não ser ideal. Nos próximos capítulos, revisitaremos essas decisões e criaremos um ecossistema mais robusto para entrega contínua da loja virtual.

Capítulo 5

PUPPET ALÉM DO BÁSICO

No final do capítulo anterior, para configurar a infraestrutura da loja virtual, o código Puppet não está muito bem organizado. A única separação que fizemos foi criar dois arquivos de manifesto, um para cada servidor: `web` e `db` . No entanto, dentro de cada arquivo, o código é simplesmente uma lista de recursos.

Além disso, os arquivos de configuração e de *templates* estão jogados sem nenhuma estrutura. Neste capítulo, vamos aprender novos conceitos e funcionalidades do Puppet ao mesmo tempo que refatoramos o código para torná-lo mais idiomático e bem organizado.

5.1 CLASSES E TIPOS DEFINIDOS

O Puppet gerencia uma única instância de cada recurso definido em um manifesto, tornando-os uma espécie de *singleton*. Da mesma forma, uma **classe** é uma coleção de recursos únicos em seu sistema.

Se você conhece linguagens orientadas a objetos, não se confunda com a terminologia. Uma classe no Puppet não pode ser instanciada diversas vezes. Classes são apenas uma forma de dar nome a uma coleção de recursos que serão aplicados como uma

unidade.

Um bom uso de classes no Puppet é para configuração de serviços que você precisa instalar apenas uma vez no sistema. Por exemplo, no arquivo `db.pp`, instalamos e configuramos o servidor MySQL, e criamos o *schema* e o usuário específicos da loja virtual. Na vida real, podemos ter vários *schemas* e usuários usando o mesmo banco de dados, porém não instalamos mais de um MySQL por sistema.

O servidor MySQL é um bom candidato para definirmos em uma classe genérica, que chamaremos de `mysql-server`. Refatorando nosso arquivo `db.pp` para declarar e usar a nova classe, teremos:

```
class mysql-server {
  exec    { "apt-update": ... }
  package { "mysql-server": ... }
  file    { "/etc/mysql/conf.d/allow_external.cnf": ... }
  service { "mysql": ... }
  exec    { "remove-anonymous-user": ... }
}

include mysql-server

exec { "loja-schema": ...,
  require => Class["mysql-server"],
}
exec { "loja-user": ... }
```

Perceba que movemos o recurso `Exec["remove-anonymous-user"]` — que remove a conta de acesso anônima — para dentro da classe `mysql-server`, pois isso é algo que deve acontecer somente quando o servidor MySQL é instalado pela primeira vez.

Outra mudança que você pode perceber é que o recurso `Exec["loja-schema"]` agora tem uma dependência com

`Class["mysql-server"]` em vez de um recurso específico `Service["mysql"]` como anteriormente. Essa é uma forma de encapsular detalhes de implementação dentro da classe, isolando esse conhecimento do resto do código, que pode declarar dependências em coisas mais abstratas e estáveis.

Para definir uma nova classe, basta escolher seu nome e colocar todos os recursos dentro de uma declaração do tipo `class <nome da classe> { ... }`. Para usar uma classe, você pode utilizar a sintaxe `include <nome da classe>` ou a versão mais parecida com a definição de um recurso com que já estamos acostumados: `class { "<nome da classe>": ... }`.

Por outro lado, os recursos que criam o *schema* e o usuário da loja virtual podem ser reaproveitados para criar *schemas* e usuários para outras aplicações rodando neste mesmo servidor. Colocá-los em uma classe seria a forma errada de encapsulamento, pois o Puppet exigiria que ela fosse única. Para situações como esta, o Puppet possui uma outra forma de encapsulamento chamada de "tipos definidos".

Um **tipo definido** (ou *defined type*) é uma coleção de recursos que pode ser usada várias vezes em um mesmo manifesto. Eles permitem que você elimine duplicação de código agrupando recursos relacionados que podem ser reutilizados em conjunto. Você pode interpretá-los como equivalentes a macros em uma linguagem de programação.

Além disso, eles podem ser parametrizados e definir valores padrão para parâmetros opcionais. Agrupando os dois recursos `exec` para criar o *schema* e o usuário de acesso ao banco de dados, em um tipo definido chamado `mysql-db`, teremos:

```
class mysql-server { ... }

define mysql-db($schema, $user = $title, $password) {
  Class['mysql-server'] -> Mysql-db[$title]

  exec { "$title-schema":
    unless  => "mysql -uroot $schema",
    command => "mysqladmin -uroot create $schema",
    path    => "/usr/bin/",
  }

  exec { "$title-user":
    unless  => "mysql -u$user -p$password $schema",
    command => "mysql -uroot -e \"GRANT ALL PRIVILEGES ON \
                                  $schema.* TO '$user'@'%' \
                                  IDENTIFIED BY '$password';\"",
    path    => "/usr/bin/",
    require => Exec["$title-schema"],
  }
}

include mysql-server

mysql-db { "loja":
  schema   => "loja_schema",
  password => "lojasecret",
}
```

A sintaxe para declarar um tipo definido é `define <nome do tipo>(<parâmetros>) { ... }`. Nesse exemplo, o tipo definido `mysql-db` aceita três parâmetros: `$schema`, `$user` e `$password`.

O parâmetro `$user`, quando não especificado, terá o mesmo valor do parâmetro especial `$title`. Para entender o valor do parâmetro `$title` — que não precisa ser declarado explicitamente —, basta olhar a sintaxe da chamada na qual declaramos uma instância do tipo definido: `mysql-db { "loja": schema => "loja_schema", ... }`.

O nome do recurso que instancia o tipo definido, neste caso `loja`, será passado como valor do parâmetro `$title`. Os outros parâmetros são passados seguindo a mesma sintaxe que usamos em outros recursos nativos do Puppet: o nome do parâmetro e seu valor separado por uma flecha `=>`.

Movemos os dois recursos `exec` que criam o *schema* e o usuário para dentro do tipo definido. Substituímos também todas as referências *hard coded* pelos respectivos parâmetros, tomando cuidado para usar aspas duplas nas strings para que o Puppet saiba expandir os valores corretamente.

Para poder usar o tipo definido mais de uma vez, precisamos parametrizar o nome dos recursos `exec` e torná-los únicos usando o parâmetro `$title`. A última modificação foi promover a declaração da dependência com a classe `mysql-server` para o topo do tipo definido. Com isso, os comandos individuais dentro do tipo definido não precisam declarar dependências com recursos externos, facilitando a manutenção desse código no futuro.

Usando classes e tipos definidos, conseguimos refatorar o código Puppet para torná-lo um pouco mais reutilizável. No entanto, não mudamos a organização dos arquivos em si. Tudo continua declarado dentro de um único arquivo. Precisamos aprender uma forma melhor de organizar nossos arquivos.

5.2 EMPACOTAMENTO E DISTRIBUIÇÃO USANDO MÓDULOS

O Puppet define um padrão para empacotar e estruturar seu código: **módulos**. Módulos possuem uma estrutura predefinida de

diretórios em que você deve colocar seus arquivos, assim como alguns padrões de nomenclatura.

Módulos também são uma forma de compartilhar código Puppet com a comunidade. O Puppet Forge (http://forge.puppetlabs.com/) é um site mantido pela Puppet Labs, no qual você pode encontrar diversos módulos escritos pela comunidade ou registrar um módulo que você escreveu e quer compartilhar.

Dependendo da sua experiência com diferentes linguagens, os equivalentes a um módulo Puppet nas comunidades Ruby, Java, Python e .NET seriam uma `gem`, um `jar`, um `egg` e uma *DLL*, respectivamente. A estrutura de diretórios simplificada para um módulo Puppet é:

```
<nome do módulo>/
├── files
│   └── ...
├── manifests
│   └── init.pp
├── templates
│   └── ...
└── tests
    └── init.pp
```

Em primeiro lugar, o nome do diretório raiz define o nome do módulo. O diretório `manifests` é o mais importante, pois lá você coloca seus arquivos de manifesto (com extensão `.pp`). Dentro dele, deve existir pelo menos um arquivo chamado `init.pp`, que serve como ponto de entrada para o módulo.

O diretório `files` contém arquivos de configuração estáticos que podem ser acessados dentro de um manifesto usando uma URL especial: `puppet:///modules/<nome do`

módulo>/<arquivo> . O diretório templates contém arquivos ERB que podem ser referenciados dentro de um manifesto usando o nome do módulo: template('<nome do módulo>/<arquivo ERB>') .

Por fim, o diretório tests contém exemplos mostrando como usar as classes e tipos definidos pelo módulo. Esses testes não fazem nenhum tipo de verificação automatizada. Você pode rodá-los usando o comando puppet apply --noop . Ele simula uma execução do Puppet sem realizar nenhuma alteração no sistema. Caso haja algum erro de sintaxe ou de compilação, você vai detectá-los na saída do comando.

Nossa próxima refatoração é uma preparação antes de criarmos o módulo. Vamos quebrar a definição da classe mysql-server e do tipo definido mysql-db em dois arquivos separados, chamados server.pp e db.pp , respectivamente. No mesmo nível em que criamos o arquivo Vagrantfile , vamos criar um novo diretório modules , onde colocaremos nossos módulos. Dentro dele, criamos um novo módulo para instalação e configuração do MySQL, chamado mysql , criando a seguinte estrutura de diretórios e movendo os respectivos arquivos para lá:

```
.
├── Vagrantfile
├── manifests
│   ├── .keystore
│   ├── context.xml
│   ├── db.pp
│   ├── devopsnapratica.war
│   ├── server.xml
│   ├── tomcat7
│   └── web.pp
└── modules
    └── mysql
        ├── manifests
```

```
│   ├── init.pp
│   ├── db.pp
│   └── server.pp
└── templates
    └── allow_ext.cnf
```

Perceba que criamos um novo arquivo chamado `init.pp` no diretório `modules/mysql/manifests`. Esse arquivo será o ponto de entrada para o módulo e ele deve declarar uma classe `mysql` vazia que serve como *namespace* para o resto do módulo:

```
class mysql { }
```

Movemos também o arquivo `allow_ext.cnf` para o diretório `modules/mysql/templates`. Por estar dentro da estrutura padrão do módulo, não precisamos mais referenciar o *template* usando um caminho absoluto. O Puppet aceita o caminho no formato `<nome do módulo>/<arquivo>` e sabe procurar o *template* `<arquivo>` dentro do módulo escolhido.

Podemos então mudar o recurso `file` no arquivo `modules/mysql/manifests/server.pp` de `template("/vagrant/manifests/allow_ext.cnf")` para `template("mysql/allow_ext.cnf")`:

```
class mysql::server {
  exec { "apt-update": ... }
  package { "mysql-server": ... }

  file { "/etc/mysql/conf.d/allow_external.cnf":
    ...
    content => template("mysql/allow_ext.cnf"),
    ...
  }

  service { "mysql": ... }
  exec { "remove-anonymous-user": ... }
}
```

Perceba que renomeamos a classe `mysql-server` para `mysql<::server` . Esta é a nomenclatura padrão que o Puppet entende para importar classes e tipos dentro do mesmo módulo. Da mesma forma, devemos também renomear o tipo definido `mysql-db` para `mysql<::db` , dentro do arquivo `modules/mysql/manifests/db.pp` , e alterar sua dependência com a classe `mysql<::server` :

```
define mysql::db($schema, $user = $title, $password) {
  Class['mysql::server'] -> Mysql::Db[$title]
  ...
}
```

Por fim, podemos simplificar bastante o arquivo `manifests/db.pp` para usar a classe `mysql<::server` e o tipo definido `mysql<::db` no novo módulo:

```
include mysql::server

mysql::db { "loja":
  schema   => "loja_schema",
  password => "lojasecret",
}
```

Com isso, criamos um módulo genérico que disponibiliza uma classe para instalar o servidor MySQL e um tipo definido para criar um *schema* e um usuário para acessá-lo. Pode parecer confuso o fato de estarmos mudando as coisas de lugar sem nenhuma mudança no comportamento, porém é exatamente por isso que refatoramos código!

Melhoramos sua estrutura sem modificar o comportamento externo. O benefício é facilitar o trabalho de manutenção no futuro, isolando componentes individuais e agrupando aquilo que é relacionado. Princípios de *design* de software — como

encapsulamento, acoplamento e coesão — também ajudam a melhorar código de infraestrutura.

Para testar que tudo continua funcionando, você pode destruir e subir a máquina virtual `db` novamente. Entretanto, antes disso, é preciso fazer uma pequena alteração para dizer ao Vagrant que temos um novo módulo que precisa estar disponível quando o Puppet executar. Mudamos o arquivo `Vagrantfile` para adicionar a configuração `module_path` nas VMs `web` e `db`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  config.vm.define :db do |db_config|
    ...
    db_config.vm.provision "puppet" do |puppet|
      puppet.module_path = "modules"
      puppet.manifest_file = "db.pp"
    end
  end

  config.vm.define :web do |web_config|
    ...
    web_config.vm.provision "puppet" do |puppet|
      puppet.module_path = "modules"
      puppet.manifest_file = "web.pp"
    end
  end

  ...
end
```

Agora que o código Puppet para gerenciar a infraestrutura do banco de dados está refatorado, é hora de melhorar o código que instala, configura e faz *deploy* da aplicação no servidor `web`.

5.3 REFATORANDO O CÓDIGO PUPPET DO

SERVIDOR WEB

Agora que sabemos como refatorar nosso código Puppet, vamos criar um novo módulo chamado `tomcat` que será usado pelo servidor web. Em primeiro lugar, vamos simplesmente criar a estrutura de diretórios do novo módulo e mover os arquivos do diretório `manifests` para os seguintes diretórios dentro de `modules/tomcat`:

```
.
├── Vagrantfile
├── manifests
│   ├── .keystore
│   ├── db.pp
│   ├── devopsnapratica.war
│   └── web.pp
└── modules
    ├── mysql
    │   └── ...
    └── tomcat
        ├── files
        │   ├── server.xml
        │   └── tomcat7
        ├── manifests
        └── templates
            └── context.xml
```

Os únicos arquivos restantes no diretório `manifests` são as definições dos servidores (`db.pp` e `web.pp`), o arquivo WAR contento a aplicação da loja virtual e o arquivo `.keystore` contendo o certificado SSL. Fizemos isso para seguir um princípio importante no desenvolvimento de software: **separação de responsabilidades**.

Se olharmos o conteúdo do arquivo `web.pp` , estamos misturando recursos mais genéricos — que instalam e configuram o Tomcat — com recursos específicos da loja virtual. Se

simplesmente movermos tudo para o mesmo módulo, ele não será reutilizável.

Pense sob o ponto de vista externo de uma outra equipe na sua empresa ou a comunidade em geral: um módulo que instala e configura o Tomcat é muito mais útil do que um módulo que configura o Tomcat com toda a loja virtual instalada.

Tendo isso em mente, o objetivo do resto da refatoração será extrair um módulo do Tomcat que não possua nenhuma referência a coisas específicas da loja virtual. Conforme formos movendo definições de recurso do arquivo `web.pp` para o módulo `tomcat`, vamos manter coisas específicas da loja virtual no manifesto do servidor e tornar os recursos do módulo `tomcat` mais genéricos.

Antes de começarmos a atacar os recursos do Tomcat, vamos começar com algo um pouco mais simples: extrair a parte que instala o cliente do MySQL para uma nova classe no módulo `mysql`. Criamos um novo arquivo `client.pp` dentro de `modules/mysql/manifests`, com o conteúdo:

```
class mysql::client {
  exec { "apt-update":
    command => "/usr/bin/apt-get update"
  }

  package { "mysql-client":
    ensure  => installed,
    require => Exec["apt-update"],
  }
}
```

Novamente, a motivação é separação de responsabilidades: a classe `mysql<::client` pode ser útil em outros contextos,

mesmo quando você não está desenvolvendo uma aplicação web. Portanto, colocá-la no módulo `mysql` faz mais sentido do que no módulo `tomcat`. Aos poucos estamos começando a desembaraçar o código.

Assim, podemos começar a mudar o arquivo `web.pp`: vamos incluir a nova classe `mysql<::client`, e remover o recurso `Exec[apt-update]` — que migrou para a nova classe. Precisamos também atualizar todas as referências aos arquivos estáticos e de *template* para o novo caminho dentro do módulo:

```
include mysql::client

package { "tomcat7": ... }

file { "/var/lib/tomcat7/conf/.keystore": ... }

file { "/var/lib/tomcat7/conf/server.xml": ...
  source  => "puppet:///modules/tomcat/server.xml", ...
}

file { "/etc/default/tomcat7": ...
  source  => "puppet:///modules/tomcat/tomcat7", ...
}

service { "tomcat7": ... }

$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"

file { "/var/lib/tomcat7/conf/context.xml": ...
  content => template("tomcat/context.xml"), ...
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

As únicas referências que devem continuar apontando para o caminho absoluto são os recursos

`File[/var/lib/tomcat7/conf/.keystore]` e `File[/var/lib/tomcat7/webapps/devopsnapratica.war]`, já que não os movemos para dentro do módulo.

Agora vamos começar a mover partes do código do arquivo `web.pp` para novas classes e tipos definidos dentro do módulo `tomcat`. Criaremos um novo arquivo `server.pp` dentro do diretório `modules/tomcat/manifests` e moveremos os seguintes recursos do arquivo `web.pp` para uma nova classe, que vamos chamar de `tomcat<::sever`:

```
class tomcat::server {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }
  service { "tomcat7": ... }
}
```

Para expor esta nova classe no módulo novo, precisamos criar também um arquivo `init.pp` dentro do mesmo diretório, contendo inicialmente uma única classe vazia: `class tomcat { }`. Com isso, podemos alterar o arquivo `web.pp`, removendo os recursos que foram movidos e substituindo-os por um `include` da nova classe `tomcat<::server`:

```
include mysql::client
include tomcat::server

file { "/var/lib/tomcat7/conf/.keystore": ... }
file { "/var/lib/tomcat7/conf/server.xml": ... }

$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"

file { "/var/lib/tomcat7/conf/context.xml": ... }
file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

Perceba que ainda não movemos todos os recursos do tipo `file` para a nova classe, pois eles possuem referências e configurações que são bastante específicas da nossa aplicação:

- O arquivo `.keystore` possui o certificado SSL da loja virtual. Não é uma boa ideia distribuir nosso certificado em um módulo genérico do Tomcat que outras pessoas ou equipes podem reutilizar.
- O arquivo `server.xml` possui configurações para habilitar HTTPS, contendo informações secretas que também são específicas da loja virtual.
- O arquivo `context.xml` configura os recursos JNDI de acesso ao *schema* do banco de dados da loja virtual.
- Por fim, o arquivo `devopsnapratica.war` contém nossa aplicação.

Para tornar o módulo do Tomcat mais genérico, vamos primeiro resolver o problema da configuração SSL e do arquivo `.keystore`. Dentro do arquivo de configuração `server.xml`, em vez de assumir que o nome do *keystore* e a senha de acesso serão sempre os mesmos, vamos transformá-los em um *template* e passar essas informações para a classe `tomcat<::server`. Para entender o resultado desejado, começaremos mostrando as alterações no arquivo `web.pp`:

```
include mysql::client

$keystore_file = "/etc/ssl/.keystore"
$ssl_connector = {
  "port"         => 8443,
  "protocol"     => "HTTP/1.1",
  "SSLEnabled"   => true,
```

```
  "maxThreads"   => 150,
  "scheme"       => "https",
  "secure"       => "true",
  "keystoreFile" => $keystore_file,
  "keystorePass" => "secret",
  "clientAuth"   => false,
  "sslProtocol"  => "SSLv3",
}
$db_host     = "192.168.33.10"
$db_schema   = "loja_schema"
$db_user     = "loja"
$db_password = "lojasecret"

file { $keystore_file:
  mode    => 0644,
  source  => "/vagrant/manifests/.keystore",
}

class { "tomcat::server":
  connectors => [$ssl_connector],
  require    => File[$keystore_file],
}

file { "/var/lib/tomcat7/conf/context.xml": ... }
file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

Vamos discutir cada uma das mudanças: em primeiro lugar, mudamos o diretório do arquivo `.keystore` para um lugar mais genérico, fora do Tomcat, e salvamos sua localização em uma nova variável `$keystore_file`. Com isso, removemos as referências e dependências com o Tomcat do recurso `File[/etc/ssl/.keystore]`.

A segunda mudança importante é que tornamos a classe `tomcat<::server` parametrizada, passando um novo parâmetro `connectors` que representa um *array* de conectores que precisam ser configurados.

Esse exemplo também serve para mostrar duas novas

estruturas de dados reconhecidas pelo Puppet: *arrays* e *hashes*. Um *array* é uma lista indexada de itens, enquanto um *hash* é uma espécie de dicionário que associa uma chave a um valor. Ao passar a lista de conectores como um parâmetro, podemos mover o recurso `File[/var/lib/tomcat7/conf/server.xml]` para dentro da classe `tomcat<::server` , no arquivo `modules/tomcat/manifests/server.pp` :

```
class tomcat::server($connectors = []) {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }

  file { "/var/lib/tomcat7/conf/server.xml":
    owner   => root,
    group   => tomcat7,
    mode    => 0644,
    content => template("tomcat/server.xml"),
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }

  service { "tomcat7": ... }
}
```

Perceba que a sintaxe de declaração da classe mudou para aceitar o novo parâmetro `$connectors` , que possui um valor padrão correspondente a uma lista vazia. Como agora o conteúdo do arquivo `server.xml` precisa ser dinâmico, precisamos movê-lo do diretório `modules/tomcat/files` para o diretório `modules/tomcat/templates` , substituindo também a definição do recurso para usar o *template* na geração do arquivo. Por fim, precisamos mudar o conteúdo do arquivo `server.xml` para configurar os conectores dinamicamente:

```
<?xml version='1.0' encoding='utf-8'?>
<Server port="8005" shutdown="SHUTDOWN">
  ...
```

```
  <Service name="Catalina">
    ...
    <Connector port="8080" protocol="HTTP/1.1"
               connectionTimeout="20000"
               URIEncoding="UTF-8"
               redirectPort="8443" />

    <%- connectors.each do |c| -%>
      <Connector
          <%= c.sort.map{|k,v| "#{k}='#{v}'"}.join(" ") %> />
    <%- end -%>
    ...
  </Service>
</Server>
```

Perceba que agora nenhum conector referencia qualquer característica específica da nossa aplicação. Usando um pouco de Ruby no *template* `server.xml` , conseguimos transformar em dados o que antes estava *hard coded*.

A parte interessante é a linha: `c.sort.map{|k,v| "#{k}='#{v}'"}.join(" ")` . A operação `sort` garante que sempre vamos iterar no *hash* na mesma ordem. A operação `map` transforma cada par de chave/valor do *hash* em uma *string* formatada como um atributo XML. Por exemplo, um hash `{"port" => "80", "scheme" => "http"}` se transformará na lista `["port='80'", "scheme='http'"]` . A operação `join` junta todas as *strings* dessa lista em uma única *string*, separando-as com um espaço. O resultado final é a *string*: `"port='80' scheme='http'"` .

Agora que resolvemos o problema da configuração SSL, podemos usar o mesmo truque para mover o recurso `File[/var/lib/tomcat7/conf/context.xml]` para dentro da classe `tomcat<::server` . Novamente começamos pela mudança desejada no arquivo `web.pp` :

```
include mysql::client
```

```
$keystore_file = "/etc/ssl/.keystore"
$ssl_conector  = ...
$db = {
  "user"     => "loja",
  "password" => "lojasecret",
  "driver"   => "com.mysql.jdbc.Driver",
  "url"      => "jdbc:mysql://192.168.33.10:3306/loja_schema",
}

file { $keystore_file: ... }

class { "tomcat::server":
  connectors   => [$ssl_conector],
  data_sources => {
    "jdbc/web"     => $db,
    "jdbc/secure"  => $db,
    "jdbc/storage" => $db,
  },
  require      => File[$keystore_file],
}

file { "/var/lib/tomcat7/webapps/devopsnapratica.war": ... }
```

Seguindo o mesmo padrão, adicionamos um novo parâmetro `data_sources` na classe `tomcat<::server` e passamos um *hash* contendo os dados de configuração de cada *data source*. Movemos também o recurso `File[/var/lib/tomcat7/conf/context.xml]` para a classe `tomcat<::server` no arquivo `modules/tomcat/manifests/server.pp`:

```
class tomcat::server($connectors = [], $data_sources = []) {
  package { "tomcat7": ... }
  file { "/etc/default/tomcat7": ... }
  file { "/var/lib/tomcat7/conf/server.xml": ... }

  file { "/var/lib/tomcat7/conf/context.xml":
    owner   => root,
    group   => tomcat7,
    mode    => 0644,
    content => template("tomcat/context.xml"),
```

```
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }

  service { "tomcat7": ... }
}
```

Precisamos também alterar o arquivo de *template* `context.xml` para gerar os recursos JNDI dinamicamente:

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
    <!-- Default set of monitored resources -->
    <WatchedResource>WEB-INF/web.xml</WatchedResource>

    <%- data_sources.sort.each do |data_source, db| -%>
      <Resource name="<%= data_source %>" auth="Container"
            type="javax.sql.DataSource" maxActive="100"
            maxIdle="30"
            maxWait="10000"
            username="<%= db['user'] %>"
            password="<%= db['password'] %>"
            driverClassName="<%= db['driver'] %>"
            url="<%= db['url'] %>"/>
    <%- end -%>
</Context>
```

Com isso conseguimos refatorar bastante nosso código, tornado-os mais organizado e mais fácil de ser reaproveitado. Nesse momento, a estrutura de diretórios com os novos módulos deve ser:

```
.
├── Vagrantfile
├── manifests
│   ├── .keystore
│   ├── db.pp
│   ├── devopsnapratica.war
│   └── web.pp
└── modules
    ├── mysql
    │   ├── manifests
```

```
│   │   ├── client.pp
│   │   ├── db.pp
│   │   ├── init.pp
│   │   └── server.pp
│   └── templates
│       └── allow_ext.cnf
└── tomcat
    ├── files
    │   └── tomcat7
    ├── manifests
    │   ├── init.pp
    │   └── server.pp
    └── templates
        ├── context.xml
        └── server.xml
```

5.4 SEPARAÇÃO DE RESPONSABILIDADES: INFRAESTRUTURA VS. APLICAÇÃO

Até agora, estamos nos forçando a escrever módulos que sejam genéricos e reaproveitáveis. No entanto, o código que restou nos manifestos `web.pp`, `db.pp` e os arquivos soltos `.keytore` e `devopsnapratica.war` ainda parecem um mau cheiro.

Uma prática comum para resolver esse problema é criar módulos específicos para a aplicação. Mas isso não contradiz os argumentos que viemos fazendo até agora?

Separar módulos de infraestrutura dos módulos de aplicação é uma forma de aplicar o princípio da separação de responsabilidades em outro nível. Essa é uma forma de composição: os módulos de aplicação ainda utilizam os módulos de infraestrutura, que continuam reaproveitáveis e independentes. A figura seguinte mostra a estrutura e as dependências desejadas entre nossos módulos.

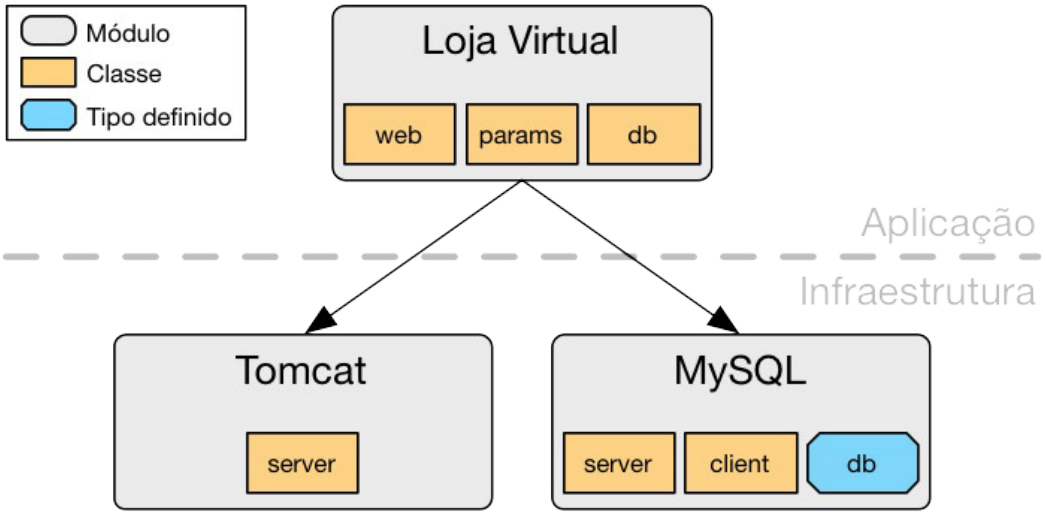

Figura 5.1: Módulos de aplicação vs. módulos de infraestrutura

O último módulo que vamos criar neste capítulo será responsável pela instalação e configuração da loja virtual. Para isso, precisamos expandir nossa estrutura de diretórios e arquivos da mesma forma que fizemos anteriormente, mas dessa vez para o novo módulo `loja_virtual`:

```
.
├── Vagrantfile
├── manifests
│   ├── db.pp
│   └── web.pp
└── modules
    ├── loja_virtual
    │   ├── files
    │   │   ├── .keystore
    │   │   └── devopsnapratica.war
    │   └── manifests
    │       ├── db.pp
    │       ├── init.pp
    │       ├── params.pp
    │       └── web.pp
    ├── mysql
    │   └── ...
    └── tomcat
        └── ...
```

Movemos os arquivos `.keystore` e `devopsnapratica.war` para o diretório `modules/loja_virtual/files` e criamos quatro arquivos de manifesto para colocarmos as novas classes do módulo `loja_virtual`: `db.pp`, `init.pp`, `params.pp` e `web.pp`. O arquivo `init.pp` simplesmente declara uma classe vazia para servir de *namespace* raiz para o resto do módulo:

```
class loja_virtual { }
```

O arquivo `db.pp` definirá uma nova classe `loja_virtual::db`, com o conteúdo movido quase sem alterações do arquivo `manifest/db.pp`:

```
class loja_virtual::db {
  include mysql::server
  include loja_virtual::params

  mysql::db { $loja_virtual::params::db['user']:
    schema   => $loja_virtual::params::db['schema'],
    password => $loja_virtual::params::db['password'],
  }
}
```

A única diferença é que movemos os valores do usuário, senha e *schema* do banco de dados para uma outra classe `loja_virtual::params` dentro do mesmo módulo. Este é um padrão comum em módulos Puppet, em que você centraliza os parâmetros em uma única classe para não precisar duplicá-los dentro do módulo.

Os parâmetros definidos na classe `loja_virtual::params` também respeitam seu *namespace*, por isso precisamos referenciá-los usando o caminho completo `$loja_virtual::params::db['user']` em vez de simplesmente `$db['user']`. Para entender melhor como esses

parâmetros foram definidos, basta olhar o conteúdo do arquivo `params.pp` que define a nova classe `loja_virtual<::params` :

```
class loja_virtual::params {
  $keystore_file = "/etc/ssl/.keystore"

  $ssl_connector = {
    "port"         => 8443,
    "protocol"     => "HTTP/1.1",
    "SSLEnabled"   => true,
    "maxThreads"   => 150,
    "scheme"       => "https",
    "secure"       => "true",
    "keystoreFile" => $keystore_file,
    "keystorePass" => "secret",
    "clientAuth"   => false,
    "sslProtocol"  => "SSLv3",
  }

  $db = {
    "user"     => "loja",
    "password" => "lojasecret",
    "schema"   => "loja_schema",
    "driver"   => "com.mysql.jdbc.Driver",
    "url"      => "jdbc:mysql://192.168.33.10:3306/",
  }
}
```

Esse é praticamente o mesmo conteúdo que tínhamos no arquivo `manifests/web.pp` . A única diferença foi a adição da chave `schema` ao *hash* `$db` , cujo valor estava anteriormente embutido no final do parâmetro `$db['url']` . Fizemos isto para poder reaproveitar a mesma estrutura nas classes `loja_virtual<::db` e `loja_virtual<::web` .

Com esta alteração, você precisa também mudar o arquivo de template `modules/tomcat/templates/context.xml` para usar a nova chave `schema` :

```
...
```

```
<Context>
  ...
  <Resource ... url="<%= db['url'] + db['schema'] %>"/>
  ...
</Context>
```

Como a definição dos parâmetros já está disponível em uma única classe, fica fácil de entender o código movido e alterado na nova classe `loja_virtual<::web` :

```
class loja_virtual::web {
  include mysql::client
  include loja_virtual::params

  file { $loja_virtual::params::keystore_file:
    mode    => 0644,
    source  => "puppet:///modules/loja_virtual/.keystore",
  }

  class { "tomcat::server":
    connectors => [$loja_virtual::params::ssl_connector],
    data_sources => {
      "jdbc/web"     => $loja_virtual::params::db,
      "jdbc/secure"  => $loja_virtual::params::db,
      "jdbc/storage" => $loja_virtual::params::db,
    },
    require => File[$loja_virtual::params::keystore_file],
  }

  file { "/var/lib/tomcat7/webapps/devopsnapratica.war":
    owner   => tomcat7,
    group   => tomcat7,
    mode    => 0644,
    source  => "puppet:///modules/loja_virtual/
                devopsnapratica.war",
    require => Package["tomcat7"],
    notify  => Service["tomcat7"],
  }
}
```

Da mesma forma que fizemos na classe `loja_virtual<::db` , incluímos a classe `loja_virtual<::params` e usamos o caminho

completo para os parâmetros. Também mudamos os recursos do tipo `file` para referenciar os arquivos usando o caminho relativo ao módulo em vez do caminho absoluto no sistema de arquivos.

Com isso, nosso novo módulo está completo e o conteúdo dos arquivos de manifesto de cada servidor fica bastante simplificado:

```
# Conteúdo do arquivo manifests/web.pp:
include loja_virtual::web

# Conteúdo do arquivo manifests/db.pp:
include loja_virtual::db
```

5.5 PUPPET FORGE: REUTILIZANDO MÓDULOS DA COMUNIDADE

Conforme o desenvolvimento de uma aplicação vai progredindo e o sistema cresce, é comum também aumentar o número de classes, pacotes e bibliotecas usados no projeto. Com isso ganhamos uma nova preocupação: **gerenciamento de dependências**.

Esse é um problema tão comum que diversas comunidades de desenvolvimento criaram ferramentas específicas para lidar com ele: na comunidade Ruby, o Bundler gerencia dependências entre `gems` ; na comunidade Java, o Maven e o Ivy gerenciam `jars` e suas dependências; até mesmo administradores de sistema usam gerenciadores de pacotes como o `dpkg` ou o `rpm` para cuidar das dependências na instalação de software.

O código de automação de infraestrutura da loja virtual também está começando a mostrar problemas no gerenciamento de suas dependências. Já temos três módulos Puppet e criamos

uma dependência indesejada entre recursos nos módulos `mysql` e `tomcat`. Se você prestar atenção, o recurso `Exec[apt-update]` é declarado duas vezes nas classes `mysql<::server` e `mysql<::client`, e é usado em três lugares diferentes: nas próprias classes e na classe `tomcat<::server`.

De modo geral, dependências são mais simples de gerenciar quando a declaração e o uso estão mais próximos. Uma dependência entre recursos declarados no mesmo arquivo é melhor do que uma dependência entre recursos declarados em arquivos diferentes; uma dependência entre arquivos diferentes do mesmo módulo é melhor do que uma dependência com recursos de módulos diferentes.

Do jeito que o código está, o módulo `tomcat` tem uma dependência direta com um recurso específico do módulo `mysql`. Se a loja virtual deixar de usar o módulo `mysql`, não será possível instalar e configurar o Tomcat. Esse tipo de dependência cria um acoplamento forte entre os módulos `mysql` e `tomcat`, tornando mais difícil utilizá-los independentemente.

Idealmente, não precisaríamos declarar nenhum recurso específico do APT. O único motivo de termos declarado esse recurso no capítulo *Infraestrutura como código* foi porque precisávamos atualizar o índice do APT (executando um `apt-get update`) antes de instalar qualquer pacote. Isso não é uma necessidade do MySQL ou do Tomcat, mas sim um problema que aparece assim que o Puppet tenta instalar o primeiro pacote no sistema.

Para melhorar nosso código e remover essa dependência indesejada, vamos aproveitar para aprender como usar módulos da

comunidade. Vamos usar um módulo da PuppetLabs que foi escrito para lidar com o APT de forma mais geral.

Ao ler a documentação do módulo (https://github.com/puppetlabs/puppetlabs-apt), vemos que ele expõe a classe `apt` e que ela possui um parâmetro `always_apt_update` que faz o que precisamos:

```
class { 'apt':
  always_apt_update => true,
}
```

Com isso, garantimos que o Puppet vai rodar o comando `apt-get update` toda vez que for executado. No entanto, precisamos garantir que o comando rode antes da instalação de qualquer pacote no sistema. Para declarar essa dependência, vamos aprender uma nova sintaxe do Puppet:

```
Class['apt'] -> Package <| |>
```

A flecha `->` impõe uma restrição na ordem de execução dos recursos. Da mesma forma que o parâmetro `require` indica uma dependência entre dois recursos, a flecha `->` garante que o recurso do lado esquerdo — nesse caso a classe `apt` — seja executado antes do recurso do lado direito.

A sintaxe `<| |>`, também conhecida como operador espaçonave, é um **coletor de recursos**. O coletor representa um grupo de recursos e é composto de: um tipo de recurso, o operador `<|`, uma expressão de busca opcional e o operador `|>`. Como deixamos a expressão de busca em branco, selecionamos todos os recursos do tipo `package`.

Juntando essas duas informações, temos um código mais

genérico que garante que o Puppet vai executar um `apt-get update` antes de tentar instalar qualquer pacote no sistema. Com isso, podemos alterar a nossa classe `loja_virtual`, declarada no arquivo `modules/loja_virtual/manifests/init.pp`:

```
class loja_virtual {
  class { 'apt':
    always_apt_update => true,
  }

  Class['apt'] -> Package <| |>
}
```

Podemos agora remover as referências ao recurso `Exec[apt-update]` dos módulos `mysql` e `tomcat`, assim como o recurso em si. O arquivo `modules/mysql/manifests/client.pp` fica bem mais simplificado:

```
class mysql::client {
  package { "mysql-client":
    ensure  => installed,
  }
}
```

Da mesma forma, o arquivo `modules/mysql/manifests/server.pp` também fica mais enxuto:

```
class mysql::server {
  package { "mysql-server":
    ensure  => installed,
  }

  file { "/etc/mysql/conf.d/allow_external.cnf": ... }
  service { "mysql": ... }
  exec { "remove-anonymous-user": ... }
}
```

Por fim, removemos a dependência indesejada na classe

`tomcat<::server`, alterando o arquivo `modules/tomcat/manifests/server.pp`:

```
class tomcat::server($connectors = [], $data_sources = []) {
  package { "tomcat7":
    ensure  => installed,
  }

  file { "/etc/default/tomcat7": ... }
  file { "/var/lib/tomcat7/conf/server.xml": ... }
  file { "/var/lib/tomcat7/conf/context.xml": ... }
  service { "tomcat7": ... }
}
```

Para completar essa refatoração, precisamos garantir que o código que adicionamos à classe `loja_virtual` seja executado nos servidores `web` e `db`. Para isso, basta incluí-lo no começo da declaração das classes `loja_virtual<::web` e `loja_virtual<::db`:

```
# Conteúdo do arquivo modules/loja_virtual/manifests/web.pp:
class loja_virtual::web {
  include loja_virtual
  ...
}

# Conteúdo do arquivo modules/loja_virtual/manifests/db.pp:
class loja_virtual::db {
  include loja_virtual
  ...
}
```

Pronto, nossa refatoração está completa. Refatorações alteram a estrutura interna do código sem alterar seu comportamento externo (FOWLER, 1999). Para garantir que nossos servidores ainda são configurados corretamente depois de todas as alterações feitas neste capítulo, vamos reprovisioná-los do zero, primeiro destruindo o servidor `db`:

```
$ vagrant destroy db
    db: Are you sure you want to destroy the 'db' VM? [y/N] y
==> db: Forcing shutdown of VM...
==> db: Destroying VM and associated drives...
==> db: Running cleanup tasks for 'puppet' provisioner...
```

Porém, ao tentar subir o servidor `db` novamente, encontramos um problema inesperado:

```
$ vagrant up db
Bringing machine 'db' up with 'virtualbox' provider...
...
Puppet::Parser::AST::Resource failed with error ArgumentError:
Could not find declared class apt at /tmp/vagrant-puppet-2/
modules-0/loja_virtual/manifests/init.pp:4 on node precise32
```

Esse erro indica que o Puppet não conseguiu encontrar a classe `apt`, pois não instalamos o módulo da PuppetLabs. A forma mais simples de instalar um módulo da comunidade é usando a própria ferramenta de linha de comando do Puppet, executando: `puppet module install <nome-do-módulo>`. Isso instalará o novo módulo e suas dependências no `modulepath` padrão, tornando-os disponíveis na próxima execução de um `puppet apply`.

Na atual configuração, não rodamos `puppet apply` diretamente. Estamos usamos o Vagrant para gerenciar a execução do Puppet. Além disso, o módulo precisa ser instalado dentro da máquina virtual, o que torna o processo de provisionamento um pouco mais complicado se quisermos executar um comando antes do Vagrant iniciar a execução do Puppet.

Conforme o número de módulos e suas dependências aumentam, fica mais difícil instalá-los um por um usando a linha de comando. A biblioteca **Librarian Puppet** (http://librarian-puppet.com/) foi criada justamente para resolver esse problema.

Você declara todas as dependências em um único arquivo `Puppetfile` e o Librarian Puppet resolve e instala os módulos nas versões especificadas em um diretório isolado. Ele funciona de forma similar ao Bundler, para quem conhece o ecossistema Ruby.

No mesmo nível em que está o arquivo `Vagrantfile`, vamos criar um novo diretório `librarian` contendo o arquivo `Puppetfile` com o seguinte conteúdo:

```
forge "http://forge.puppetlabs.com"

mod "puppetlabs/apt", "1.4.0"
```

A primeira linha declara que usaremos o PuppetLabs Forge como fonte oficial para resolver e baixar os módulos. A segunda linha declara uma dependência com o módulo `pupppetlabs/apt` na versão `1.4.0`.

Para executar o Librarian Puppet antes de usar o Puppet, vamos instalar um novo *plugin* do Vagrant. *Plugins* são uma forma de estender o comportamento básico do Vagrant, permitindo que a comunidade escreva e compartilhe diversas melhorias. Outros exemplos de *plugins* do Vagrant são: suporte a outras ferramentas de virtualização, máquinas virtuais na nuvem etc.

Para instalar o *plugin* do Librarian Puppet (https://github.com/mhahn/vagrant-librarian-puppet/), basta executar:

```
$ vagrant plugin install vagrant-librarian-puppet
Installing the 'vagrant-librarian-puppet' plugin. This can
  take a few minutes...
Installed the plugin 'vagrant-librarian-puppet (0.6.0)'!
```

Como estamos gerenciando os módulos do Librarian Puppet

em um diretório isolado, precisamos configurá-lo no arquivo `Vagrantfile` e adicioná-lo no `modulepath`:

```
VAGRANTFILE_API_VERSION = "2"

Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  config.librarian_puppet.puppetfile_dir = "librarian"

  config.vm.define :db do |db_config|
    ...
    db_config.vm.provision "puppet" do |puppet|
      puppet.module_path = ["modules", "librarian/modules"]
      ...
    end
  end

  config.vm.define :web do |web_config|
    ...
    web_config.vm.provision "puppet" do |puppet|
      puppet.module_path = ["modules", "librarian/modules"]
      ...
    end
  end
  ...
end
```

Com isso, podemos tentar provisionar a máquina virtual `db` novamente. Dessa vez, se tudo der certo, veremos que nosso código de automação de infraestrutura continua funcionando mesmo após todas as refatorações feitas até aqui:

```
$ vagrant reload db
==> db: Checking if box 'hashicorp/precise32' is up to date...
...
$ vagrant provision db
==> db: Installing Puppet modules with Librarian-Puppet...
==> db: Running provisioner: puppet...
Running Puppet with db.pp...
...
notice: Finished catalog run in 56.78 seconds
```

5.6 CONCLUSÃO

Missão cumprida! Comparado com o começo do capítulo, não apenas melhoramos a estrutura e a organização do nosso código como também aprendemos novas funcionalidades e características do Puppet. Os arquivos que estavam grandes e confusos terminaram o capítulo com apenas uma linha de código cada.

Agora os arquivos de manifesto representam o que será instalado em cada servidor. Aprendemos também como reutilizar módulos da comunidade e gerenciar dependências utilizando ferramentas de código livre, como o Librarian Puppet e o respectivo *plugin* do Vagrant.

Como um exercício para reforçar seu entendimento, imagine se quiséssemos instalar a loja virtual completa em um único servidor: quão difícil seria esta tarefa agora que temos módulos bem definidos?

Um outro exercício interessante é usar o que aprendemos sobre o Puppet para automatizar a instalação e configuração do servidor de monitoramento do capítulo *Monitoramento*.

Mas é claro que ainda há espaço para melhoria. Escrever código limpo e de fácil manutenção é mais difícil do que simplesmente escrever código que funciona. Precisamos refletir e decidir quando o código está bom o suficiente para seguir em frente.

Por enquanto, seguindo o espírito de melhoria contínua, decidimos parar de refatorar mesmo sabendo que ainda existem maus cheiros no nosso código. Em particular, entregar um arquivo

WAR estático como parte de um módulo Puppet só funciona se você nunca precisar alterá-lo.

Sabemos que isso não é verdade na maioria das situações, então vamos revisitar essa decisão nos próximos capítulos. Mas antes disso, precisamos aprender uma forma segura e confiável de gerar um novo arquivo WAR toda vez que fizermos uma alteração no código da loja virtual.