Contents

CHAPTER

9

Infrastructure as Code

To make error is human. To propagate error to all server in automatic way is #devops.

—DevOps Borat[1]

Recent years have seen the rise of disciplines like continuous integration, test driven development, build/deployment automation, and more. All of these had a purpose to automate as many parts as possible of the lifecycle of a software product. However, the main focus often is the software itself, and the infrastructure on which the software runs is still quite often a "work of art."

In a classic sense, infrastructure summarizes items such as operating systems, servers, switches, and routers (see Chapter3). According to other definitions, infrastructure comprises all of the environments of an organization together with supporting services, such as firewalls and monitoring services (see Chapter 8). The use of the term *infrastructure as code* is widespread these days. In this context, infrastructure is often thought to include every part of the solution that is not the developed software application itself (although even the software is included occasionally). In that sense, infrastructure is meant to include the middleware (such as web servers with configuration files, software packages as part of the operating system, crontabs, technical users, and so on).

Infrastructure is set up and changed over time, before the software even goes into production. If you're lucky, infrastructure is documented well, but often enough it would not be an easy task to rebuild your infrastructure from scratch if the need would arise.

"Perl was designed as a programming language for automating system administration. It didn't take long for leading-edge sysadmins to realize that handcrafted configurations and non-reproducible incantations were a bad way to run their shops."[2]

---

[1] `http://twitter.com/#!/devops_borat/status/41587168870797312`.
[2] See Mike Loukides, *What Is DevOps?* (O'Reilly, 2012), `http://radar.oreilly.com/2012/06/what-is-devops.html`.

Infrastructure as code has lately become popular to emphasize the need to handle the set up of your infrastructure in the same way you would handle development of your code: pick the right language or tool to do the job and start developing a solution that suits your needs, making it an executable specification that can be applied to target systems efficiently and repeatedly.

This chapter introduces the tools Vagrant,[3] a tool for managing virtualized development environments, and Puppet,[4] a tool for managing infrastructure items that are often also called configurations. Afterward, we'll explore a real-world use that is based on how the development of the build server Jenkins is streamlined by Vagrant and Puppet.

Starting with Infrastructure as Code

Setup and maintenance of infrastructure were automated even before the rise of Agile software development and the DevOps movement. But there have often been handcrafted, scripted solutions, barely readable by someone other than the original author. In recent years, a few tools in the field of configuration management started to gain popularity to address these challenges. These tools help developers and operations to work together and enable more transparency on the infrastructure level. After all, in today's world of ever more complex and distributed IT systems, there's an increasing need for developers to know about operational things and vice versa. The infrastructure as code paradigm and its related tools can help to achieve this goal.

Before we get into more detail, there are two key questions to be addressed:

- Why should I adopt infrastructure as code?
- How should I do it? (which tools can I use?)

To answer these questions, let's first think about the process involved in infrastructure set up when a typical web application gets developed.

Traditional Infrastructure Handling

In the first phase, as the architecture is not yet fixed, developers will try components, eventually ending up in a first draft of the setup. Now, each developer has set up his or her local development environment, with all the components running on the machine. At this point there might not even be a shared environment on which the software gets deployed continuously. That would be the next step. Depending on the organization's structure, this might already be the borderline where the developers are not allowed full access to the target machines anymore and they have to provide the operations team with some sort of assistance on how to get things up. The same might happen again for QA, staging, and production environments. And as reality goes, at least the configuration of the components might change continuously along the way, if not new components being added to the infrastructure.

[3] `http://vagrantup.com`.
[4] `http://puppetlabs.com`.

Each time developers try things on their local or test environments, they'll then report the change in infrastructure to operations, who then adjust the other environments. You might say this is nonsense if you are working in a startup or a young and small company. But unfortunately this is often the hard reality in larger corporations and, obviously, there is a lot of potential for errors and misunderstandings. Both options, no documentation and documenting infrastructure in Office documents, are suboptimal.

Furthermore the same problems can arise if new developers are added to the team. If the setup is not fully documented or someone made an effort to write a script to at least do parts of the set up, or it is documented but outdated, the new developer might struggle to get everything running smoothly, and finding the error can be a search for the needle in the haystack. There is no central, versionized version of the infrastructure that can be considered to be always runable.

How to Do It Better

Now, let's think about what would be better for everyone involved. Assume that new developers don't have to set up things by hand or partly aided by some scripts (not to mention the potential that the script might not be suitable for the developers operating system or the fact that the developer already has other stuff running his or her machine, which can interfere with the project setup), but can have them writing executable specifications, utilizing virtualization to create and destroy their test environments on the fly, as needed.

Getting new developers on board would simply be a matter of having them check out the specification from the version control repository, execute it, and use the same environment as their colleagues. If the team decides that changes to the infrastructure have reached a stable level, the specification can be used to update any shared environments (e.g., for acceptance or integration tests) and can be passed over to the operations team, who can reuse the specification for staging and production environments. Sounds great, doesn't it?

Well, within the past few years, a number of tools have emerged that address exactly these problems. Some tools existed before terms such as DevOps and infrastructure as code were coined, but these movements helped to further spread the use of these tools and grow the whole community around it.

In this chapter, I'm going to introduce two tools: Vagrant and Puppet. Vagrant allows you to easily build lightweight and portable virtual environments, based on a simple textual description. Puppet is a configuration management tool that uses a declarative syntax to describe the desired state of a target environment and allows this description to be executed to create that state on a target machine. In combination, this can lead to a target topology, similar to that shown in Figure 9-1.

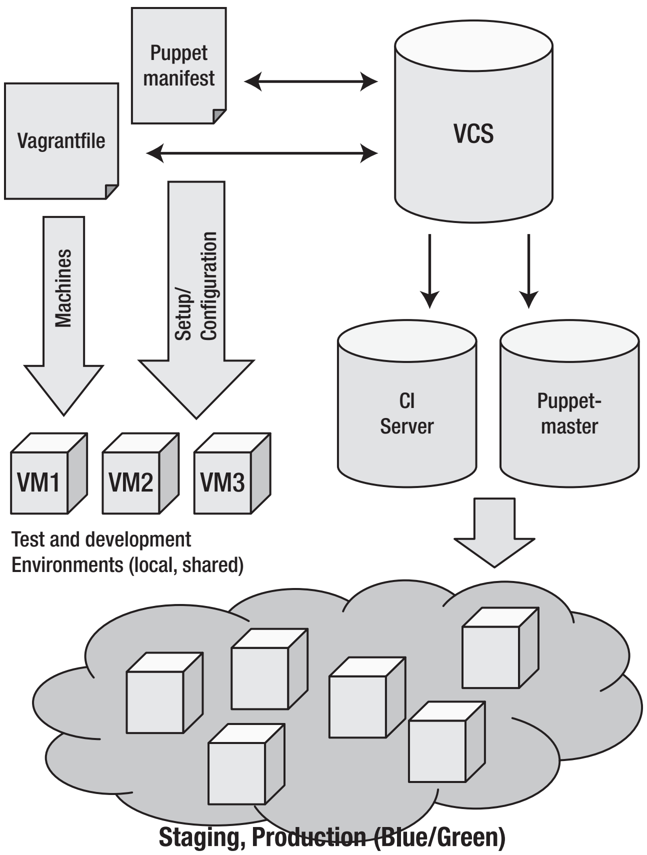


Figure 9-1. *An example topology for infrastructure as code consisting of Vagrant and Puppet artifacts that are stored in a version control system, built continuously with a continuous integration (CI) server.*

In the middle of the solution, Vagrant is used to set up test and development environments (as virtual machines). Puppet is used to provision infrastructure.

Configuration files for both Vagrant and Puppet are versionized in a version control system, with all its benefits, including change control and sharing changes in the whole team. A continuous integration (CI) server, such as Jenkins, listens to changes in version control and could propagate new versions to target environments for test purposes, automatically. Sounds like magic? It is, but don't let it scare you. We'll examine this in much more detail in this chapter. Now let's start with test environments with Vagrant.

Test Environments with Vagrant

Vagrant allows you to build virtual environments in an easy way, based on a textual specification in the so-called Vagrant file. This file is all that is needed to create a virtual environment from scratch.

VIRTUAL ENVIRONMENT

The term *virtual environment* describes a means of delivering computing resources that are independent from physical machines. A virtual environment can enable the running of virtual desktops, servers, or other virtual appliances. The advantage of a virtual environment is that it can more efficiently utilize physical resources while avoiding costly overprovisioning.

Vagrant is based on Ruby[5] and uses Oracles VirtualBox[6] to run virtual machines, so you'll need these before going on with the installation. The easiest way is then to install Vagrant via RubyGems:

```
> sudo gem install vagrant
```

If everything went well, you should see output similar to this:

```
Fetching: vagrant-1.0.2.gem (100%)
Successfully installed vagrant-1.0.2
1 gem installed
Installing ri documentation for vagrant-1.0.2...
Installing RDoc documentation for vagrant-1.0.2...
```

Now you can go on and set up your very first Vagrant environment:

```
> mkdir -p vagrant/test
> cd vagrant/test
> vagrant box add lucid32 http://files.vagrantup.com/lucid32.box
> vagrant init lucid32
```

The Add command will download a Vagrant base box from the given location. Add it to the system under the given alias and store it in `$HOME/.vagrant`. Base boxes are Vagrant's initial building blocks to create virtual environments. To make life simple here, we stick with the `Ubuntu 10.4` box that is offered by the Vagrant team. The second command creates the Vagrant

[5] http://www.ruby-lang.org/en.
[6] https://www.virtualbox.org.

file in the current directory. This initial Vagrant file contains a lot of documentation on several options available, so it's worth taking a look at it. The first lines should look like those shown in Listing 9-1.

Listing 9-1. *An Initial Vagrant File*

```
# Every Vagrant virtual environment requires a box to build
config.vm.box = "lucid32"
# The url from where the 'config.vm.box' box will be
# fetched if it doesn't already exist on the user's system.
# config.vm.box_url = "http://domain.com/path/to/above.box"
End
```

The only uncommented option is the name of the base box to build the environment from. This is the `Ubuntu` box we just added to the system. You can see that there is also the possibility to add a URL to specify where the base box can be downloaded from. Especially if you think you will use Vagrant in a team, this is an important option to make your Vagrant files portable. If the Vagrant file gets used by someone who doesn't have the base box on his or her system already, Vagrant will download it.

A VAGRANT BOX

A "box" is the base image used to create a virtual environment with Vagrant. It is a portable file that can be used by others on any platform that Vagrant runs in order to bring up a running virtual environment.

For the very first step, that is all that's needed. Although your virtual `Ubuntu` box right now, you can already bring it up by typing:

```
> vagrant up
```

This should result in an output similar to that shown in Listing 9-2.

Listing 9-2. *An Example Output After Starting Vagrant*

```
[default] Importing base box 'lucid32'...
[default] The guest additions on this VM do not match the
install version of VirtualBox! This may cause things such as
forwarded ports, shared folders, and more tonot work properly.
If any of those things fail on this machine, please update the
guest additions and repackage the box.

Guest Additions Version: 4.1.0
VirtualBox Version: 4.1.10
```

```
[default] Matching MAC address for NAT networking...
[default] Clearing any previously set forwarded ports...
[default] Forwarding ports...
[default] -- 22 => 2222 (adapter 1)
[default] Creating shared folders metadata...
[default] Clearing any previously set network interfaces...
[default] Booting VM...
[default] Waiting for VM to boot. This can take a few minutes.
[default] VM booted and ready for use!
[default] Mounting shared folders...
[default] -- v-root: /vagrant
```

You can see that Vagrant first imports and checks the base box (ignore the warning for the moment), sets up port forwarding and shared folders, and then boots the virtual machine. The folder with the Vagrant file gets mounted as a shared folder in the virtual machine under /vagrant.

Congratulations, you've successfully created your first Vagrant virtual environment! Now you likely want to log in to it and do some interesting stuff. As we saw in the log output in Listing 9-2, Vagrant created a port forwarding from the guest (the virtual machine, VM in short) port 22 to the host (your machine) port 2222, so that is an option to log in with Vagrant's default account (every Vagrant box has a user vagrant with the password vagrant). The more standard way to log in to any Vagrant box is to type:

```
> vagrant ssh
```

in the command line and you will be logged in.

With the default setup, your box is not visible to the outside, except the default port forwarding. However, to use it as a test system, you'll need more components to be accessible to you. One way to achieve this is to define your own port forwarding. Let's assume we want to set up a web server on the box and make it available via the host port 8001. That would look like this in the Vagrant file:

```
Vagrant::Config.run do |config|
config.vm.box = "lucid32"
config.vm.forward_port 80, 8001
end
```

To make this change in your configuration visible, reload your Vagrant environment:

```
> vagrant reload
```

This will shut down and reboot the box with the new configuration. Now, log in to your box, install Apache, and you'll be able to see the web server default page at http://localhost:8001.

```
> vagrant shh
lucid32> sudo apt-get update
lucid32> sudo apt-get install apache2
lucid32> nano /var/www/index.html
```

Finally, if you don't need your environment anymore, you can halt, suspend, or destroy it:

```
> vagrant halt|suspend|destroy
```

Halting and suspending preserves the state of your VM (i.e., the underlying VirtualBox VM will not be removed). If you destroy a box, all changes you made will be lost.

Host-Only Networking, Multi-VM Environments

You might already have thought that setting up port forwarding for every component on your test system is not really efficient, and you're right. Besides that, one of our initial goals was to create production-like environments in order to eliminate configuration errors early. Speaking of production-like, it is not realistic to believe that all your services will be running on one host only, so it would also be nice to handle several VMs with one Vagrant file. Fortunately, Vagrant addresses all these issues in the form of host-only networking and multi-VM environments.

As the name indicates, multi-VM environments let you handle several boxes with one Vagrant file. If you define more than one box in Vagrant, you need to add the name of the box to the Vagrant commands in order to let it know which box to boot, destroy, ssh in, and so forth. Most of the commands work without a name and will then be applied to all boxes defined in the current Vagrant file.

The other feature, host-only networking, enables you to assign static IP addresses to your VMs that are only accessible from the host machine. As long as you do not configure separate netmasks for the boxes, they can also see each other.

Listing 9-3 shows an example for both of these features.

Listing 9-3. *Example for Host-Only Networking, Multi-VM Environments*

```
Vagrant::Config.run do |config|
   config.vm.box = "lucid32"
   config.vm.box_url = "http://files.vagrantup.com/lucid32.box"
   config.vm.define :web do |web|
      web.vm.network :hostonly, "33.33.33.11"
   end
   config.vm.define :db do |db|
      db.vm.network :hostonly, "33.33.33.12"
   end
end
```

Running the Up command creates two VMs. You can log in to them with:

```
> vagrant ssh web|db
```

Both virtual machines are reachable from your computer under the specified IP addresses.

Provisioning with Puppet

As we have now covered the basics, we can finally get to the point this is all about: handling the set up of your infrastructure as code. Up until now, we only created some lightweight virtual machines. That's handy, but ultimately, we want to create machines complete with software and configuration.

As Vagrant also offers a mechanism to configure provisioning, we'll stick to it to introduce the next tool, Puppet. As mentioned before, Puppet is a configuration management tool, based on Ruby. It allows you to create so-called manifests, which include a description of the system in question.

Vagrant also allows you to use other provisioning tools, like Chef,[7] or do provisioning with shell scripts, but we will concentrate on Puppet here.

Let's get back to the small Apache web server example we used above to check that our VM was up and running and rebuild it with the help of Puppet. First, we need to tell Vagrant that it should use Puppet for provisioning and where it can find the Puppet manifest, as shown in Listing 9-4:

Listing 9-4. *Configure Puppet in Vagrant*

```
Vagrant::Config.run do |config|
  config.vm.box = "lucid32"
  config.vm.box_url = "http://files.vagrantup.com/lucid32.box"
  config.vm.network :hostonly, "33.33.33.10"
  config.vm.provision :puppet do |puppet|
  puppet.manifests_path = "manifests"
  puppet.manifest_file  = "webserver.pp"
end
```

We tell Vagrant to look for the Puppet manifests in the manifests folder and to use the file `webserver.pp` there. Before we take a closer look at the content of this manifest, let's quickly look at how Puppet works. Puppet uses a domain-specific language to describe a system in the form of resources. A resource can be nearly anything, from plain files, to software packages, services, or even command executions. These resources can then be grouped together in classes, modules, and node definitions. Visit the excellent online documentation for Puppet to learn more.

This is what a Puppet resource looks like:

```
<resource type> { "resource-name":
 attribute_name => attribute_value,
 ...
}
```

It consists of a certain resource type (e.g., file, group), a name, and a series of attributes in the form of key-value pairs. With these basics in mind, let's take a look at our web server manifest, as shown in Listing 9-5:

Listing 9-5. *Puppet Manifest Configuring the Web Server*

```
# to fix missing puppet group in lucid32 box
group { "puppet":
 ensure => present,
}

# to update outdated package list
exec { "refresh-packages":
```

[7] http://www.opscode.com/chef.

```
 command => "/usr/bin/apt-get update",
 before => Package["apache2"],
}

package { "apache2":
 ensure => installed,
}

service { "apache2":
 ensure => running,
 require => Package["apache2"],
}

file { "/var/www/index.html":
 ensure => file,
 content => "<html><body><h1>Vagrant and Puppet rocks!</h1></body></html>",
 require => Service["apache2"],
}
```

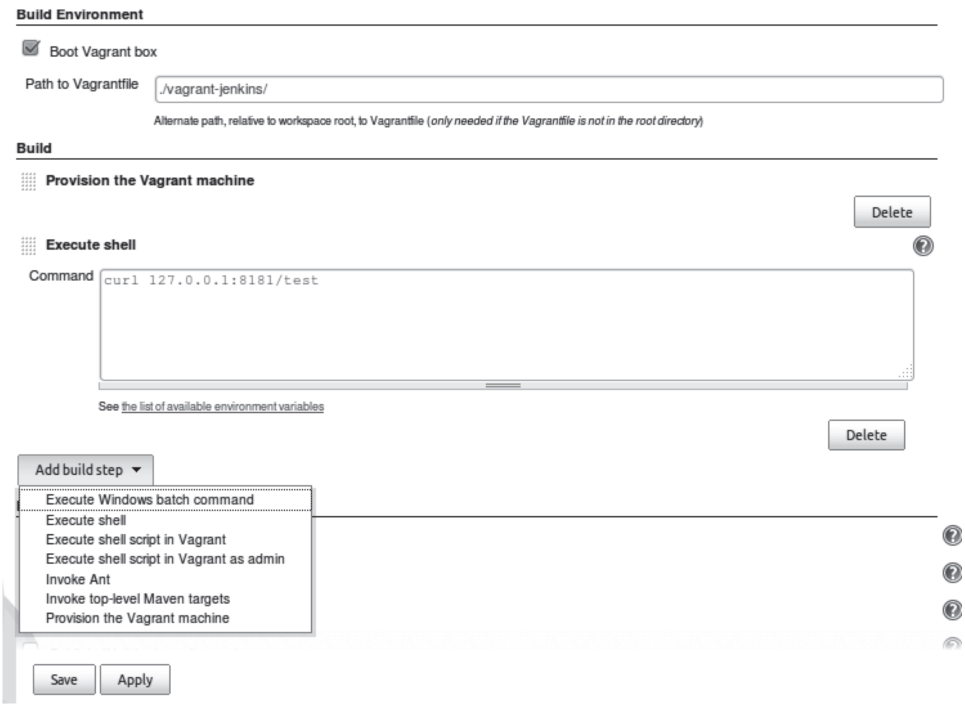
The first resource states that a group with the name `puppet` should be present on the system in question. If this is not the case, Puppet will go on and create that group. The ensure parameter can be found on many Puppet resources to specify the desired state for the resource. The next resource is a command execution to refresh the `Ubuntu` package repository list. The next two resources tell Puppet that the operating system's underlying package manager should install the `'apache2'` package if it is not installed yet and that the related service should be running. Finally, the last resource alters the content of the default Apache index page.

You might notice that some resources have parameters like before and require the so-called metaparameters. They indicate dependencies between resources for Puppet, as it has by itself no guaranteed execution order for its resources. The dependency between the package and the service is typical, because obviously the package needs to be installed before Puppet can check if the related service is up and running.

If you start this virtual machine now, you'll see some more output in contrast to running Vagrant without provisioning, and after everything is finished, you can go to `http://33.33.33.10` and see the modified index page.

Using Vagrant for Continuous Integration

While reading through the above, you might already have thought about integrating these tools into your continuous integration setup to create test environments on the fly. If you use the popular Jenkins CI server, there's already a plug-in, written by Tyler Croy. It adds new build steps to Jenkins to run and provision Vagrant boxes during build jobs. Just use Jenkins plug-in management page to install the plug-in, then create a new job and configure it as shown in Figure 9-2.

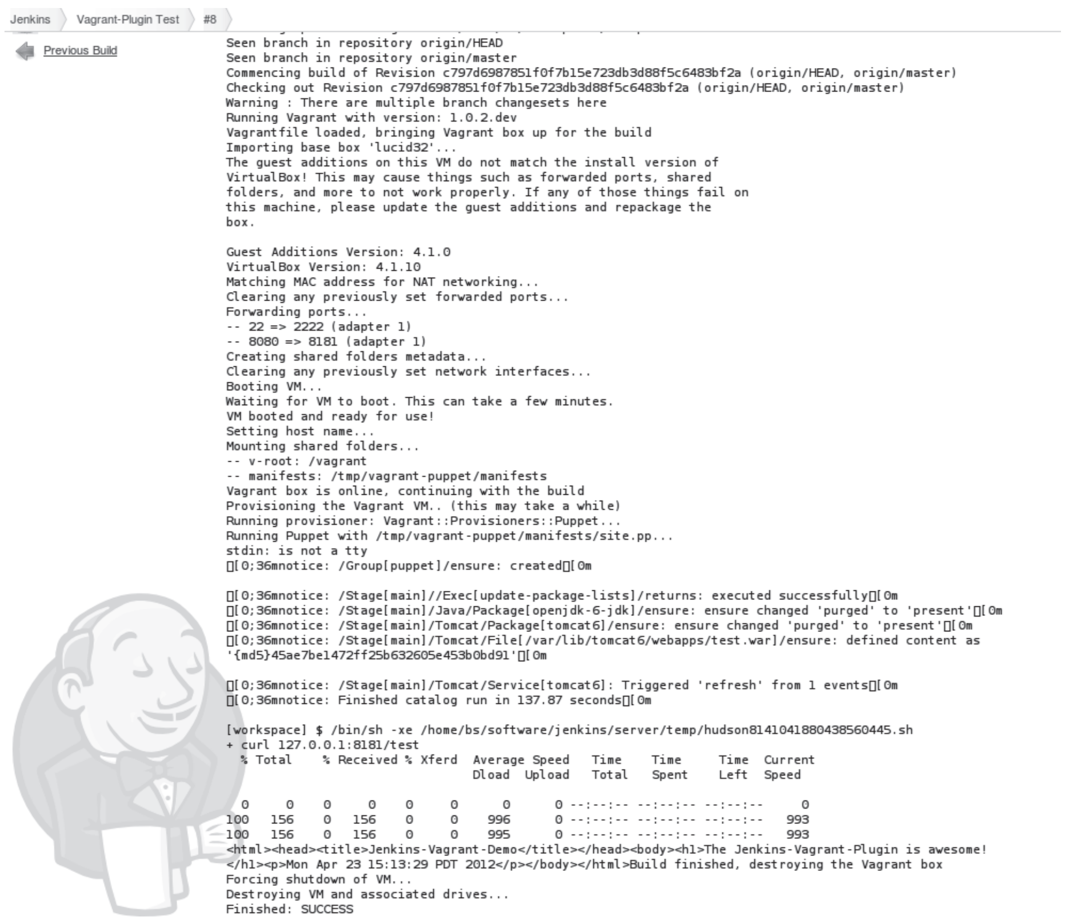

Build Environment
☑ Boot Vagrant box
Path to Vagrantfile ./vagrant-jenkins/
Alternate path, relative to workspace root, to Vagrantfile (only needed if the Vagrantfile is not in the root directory)
Build
Provision the Vagrant machine
Delete
Execute shell
Command
See the list of available environment variables
Delete
Add build step
Execute Windows batch command
Execute shell
Execute shell script in Vagrant
Execute shell script in Vagrant as admin
Invoke Ant
Invoke top-level Maven targets
Provision the Vagrant machine
Save
Apply

```
curl 127.0.0.1:8181/test
```

Figure 9-2. *Configuration of Vagrant as Jenkins Build Job*

You can see we've ticked the box to tell Jenkins to run Vagrant during the job and told it where to find the Vagrant file. Then, in the build itself, the provisioning step of Vagrant is executed (it isn't while booting the box here) and finally, we use cURL, a command line tool for transferring data (see `http://curl.haxx.se`), to check if the web server is up and running and serves our demo index page.

If you run this job and go to the console output in Jenkins, you can see all of the above steps executed, as shown in Figure 9-3.



Jenkins > Vagrant-Plugin Test > #8

Previous Build

```
Seen branch in repository origin/HEAD
Seen branch in repository origin/master
Commencing build of Revision c797d6987851f0f7b15e723db3d88f5c6483bf2a (origin/HEAD, origin/master)
Checking out Revision c797d6987851f0f7b15e723db3d88f5c6483bf2a (origin/HEAD, origin/master)
Warning : There are multiple branch changesets here
Running Vagrant with version: 1.0.2.dev
Vagrantfile loaded, bringing Vagrant box up for the build
Importing base box 'lucid32'...
The guest additions on this VM do not match the install version of
VirtualBox! This may cause things such as forwarded ports, shared
folders, and more to not work properly. If any of those things fail on
this machine, please update the guest additions and repackage the
box.

Guest Additions Version: 4.1.0
VirtualBox Version: 4.1.10
Matching MAC address for NAT networking...
Clearing any previously set forwarded ports...
Forwarding ports...
-- 22 => 2222 (adapter 1)
-- 8080 => 8181 (adapter 1)
Creating shared folders metadata...
Clearing any previously set network interfaces...
Booting VM...
Waiting for VM to boot. This can take a few minutes.
VM booted and ready for use!
Setting host name...
Mounting shared folders...
-- v-root: /vagrant
-- manifests: /tmp/vagrant-puppet/manifests
Vagrant box is online, continuing with the build
Provisioning the Vagrant VM.. (this may take a while)
Running provisioner: Vagrant::Provisioners::Puppet...
Running Puppet with /tmp/vagrant-puppet/manifests/site.pp...
stdin: is not a tty
[0;36mnotice: /Group[puppet]/ensure: created[0m

[0;36mnotice: /Stage[main]//Exec[update-package-lists]/returns: executed successfully[0m
[0;36mnotice: /Stage[main]/Java/Package[openjdk-6-jdk]/ensure: ensure changed 'purged' to 'present'[0m
[0;36mnotice: /Stage[main]/Tomcat/Package[tomcat6]/ensure: ensure changed 'purged' to 'present'[0m
[0;36mnotice: /Stage[main]/Tomcat/File[/var/lib/tomcat6/webapps/test.war]/ensure: defined content as
'{md5}45ae7be1472ff25b632605e453b0bd91'[0m

[0;36mnotice: /Stage[main]/Tomcat/Service[tomcat6]: Triggered 'refresh' from 1 events[0m
[0;36mnotice: Finished catalog run in 137.87 seconds[0m

[workspace] $ /bin/sh -xe /home/bs/software/jenkins/server/temp/hudson8141041880438560445.sh
+ curl 127.0.0.1:8181/test
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed

  0     0    0     0    0     0      0      0 --:--:-- --:--:-- --:--:--     0
100   156    0   156    0     0    996      0 --:--:-- --:--:-- --:--:--   993
100   156    0   156    0     0    995      0 --:--:-- --:--:-- --:--:--   993
<html><head><title>Jenkins-Vagrant-Demo</title></head><body><h1>The Jenkins-Vagrant-Plugin is awesome!
</h1><p>Mon Apr 23 15:13:29 PDT 2012</p></body></html>Build finished, destroying the Vagrant box
Forcing shutdown of VM...
Destroying VM and associated drives...
Finished: SUCCESS
```

Figure 9-3. *Jenkins Build Job Output*

In a real-world scenario, you would likely mix several projects in this build: the one holding your infrastructure specification and the Vagrant file, the application you want to test, maybe another project that includes some sort of deployment automation (e.g., with Maven and the cargo-plug-in), and finally some sort of acceptance or integration test project that would get executed against the virtual environment.

With this type of setup, differences in configuration for all the environments can be found early and get eliminated before they cause problems later on.

At the time of writing, the Vagrant plug-in for Jenkins is still in an early stage of development. In tests, host-only networks didn't work reliably at this early moment in its development, but nonetheless the plug-in can already be very helpful to give faster feedback to your developers and system administrators. Also, as it is open source, just go over to GitHub[8] and start contributing if you find any bugs.

Complementary Tools

What else is there to say about Vagrant? Until now, we only used the `Ubuntu` base box that is available from the Vagrant web site, but it is likely you want to use other operating systems to set up your own environments. There is documentation on how to build your own boxes with VirtualBox and package them for later reuse on the Vagrant home page. Another option is to check out `http://vagrantbox.es`, a web site that lists boxes (i.e., templates for different operating systems) that have been built by members of the Vagrant community and posted for download.

A third option is to use the open source tool Veewee,[9] which makes building new base boxes for Vagrant even easier. Since version 1.0, Vagrant also offers support for writing plug-ins, and it remains to be seen what will evolve there. Check out the Vagrant page for more information (see `http://vagrantup.com/`). There are also links to an Internet relay chat channel and a Google group, where you'll find like-minded people willing to help.

You can edit Puppet manifests and modules with a simple text editor, but to have autocompletion and other nice features you may prefer to use Geppetto,[10] which is build on Eclipse. You can check that Puppet manifests conform to the style guide with puppet-lint.[11] Another helpful tool is guard-puppet,[12] which helps to reapply Puppet configs automatically.

Tools like Cucumber-Puppet[13] and puppet-rspec[14] help to test your manifests. They let you write Cucumber and RSpec tests for your manifests in order to make sure all of them comply to your manifest policies. With puppet-rspec, tests follow the following structure:

```
require 'spec_helper'
describe '<name of the thing being tested>' do
  # Your tests go in here
end
```

[8] `https://github.com/`.
[9] `https://github.com/jedi4ever/veewee`.
[10] `https://github.com/cloudsmith/geppetto`.
[11] `https://github.com/rodjek/puppet-lint`.
[12] `http://rubygems.org/gems/guard-puppet`.
[13] `http://projects.puppetlabs.com/projects/cucumber-puppet`.
[14] `https://github.com/rodjek/rspec-puppet`.

All these neat tools help to apply the practice of test-driven development (TDD)[15] to Puppet manifests and to add Puppet manifests to your continuous integration and continuous delivery system.

Provisioning with Puppet

We saw the benefits of using Puppet to manage virtual environments together with Vagrant. This is already quite useful for development teams and also can be used to set up and tear down QA environments as needed. But there is a lot more to Puppet, and its actual role is to manage every type of environment, be it development or production. Let's take a look at how to set up a Puppet server that stores the configuration for all of its managed nodes in a central place and distributes them as needed.

Setting Up a Puppet Master

To get started, we first need to install Puppet on the local system, which will be our Puppet master. This is the host that runs Puppet in the server mode and listens for Puppet agents to pull their configuration from there. In Ubuntu, the installation is quite simple:

```
> sudo apt-get install puppet
```

This installs Puppet on your local machine and puts configuration files into /etc/puppet. As I aim to make the examples simple, we enable autosigning of new Puppet clients, to avoid signing and exchanging certificates between the Puppet master and the clients. In a production environment you should not do this. To enable autosigning, put a file autosign.conf into /etc/puppet, containing the following content:

```
*.example.com
```

This will automatically sign all certificate requests from clients with a matching hostname (e.g., web01.example.com) and remove the extra step of doing a certificate exchange before master and client can communicate with each other. Now, restart the Puppet master to make our change visible:

```
> sudo /etc/init.d/puppetmaster restart
```

Setting Up a Puppet Client

Now we're ready to set up a host that acts as a Puppet client. Again, we use Vagrant to set up a host. Just create a new Vagrant file, with the content shown in Listing 9-6:

Listing 9-6. *Configure a Puppet Client*

```
Vagrant::Config.run do |config|
 config.vm.box = "lucid32"
 config.vm.box_url = "http://files.vagrantup.com/lucid32.box"
```

[15] http://www.jedi.be/blog/2011/12/13/testdriven-infrastructure-with-vagrant-puppet-guard/.

```
 config.vm.network :hostonly, "33.33.33.33"
 config.vm.host_name = "node01.example.com"
end
```

Boot this box and ssh into it. To finally enable it to connect to your Puppet master, we have to add an entry with the Puppet master's IP address and hostname to the box /etc/hosts. Additionally, we need to add a group 'puppet' to the system, which is missing in the lucid32 box. Now your virtual host can run the Puppet agent. Normally the agent would daemonize and run periodically in the background to ask the master for up-to-date configuration. To illustrate what happens behind the scene, we're going to run things by hand here.

```
> puppet agent --test --server <puppetmaster hostname>
```

This should result in output similar to Listing 9-7:

Listing 9-7. *Output of Testing the Puppet Agent*

```
info: Creating a new SSL key for node01.example.com
warning: peer certificate won't be verified in this SSL session
info: Caching certificate for ca
warning: peer certificate won't be verified in this SSL session
info: Creating a new SSL certificate request for
node01.example.com
info: Certificate Request fingerprint (md5):
1F:7D:07:3E:FC:0E:8F:67:18:A6:DC:8D:DE:E7:4A:2E
warning: peer certificate won't be verified in this SSL session
info: Caching certificate for node01.example.com
info: Caching certificate_revocation_list for ca
err: Could not retrieve catalog from remote server: Error 400 on
SERVER: Could not find default node or by name with
'node01.example.com, node01.example, node01' on node
node01.example.com
warning: Not using cache on failed catalog
err: Could not retrieve catalog; skipping run
```

You see the agent and master exchange certificates and that the master tries to find a catalog for the requesting node. As we haven't yet configured anything, the agent doesn't receive any information and aborts the run. If the agent would be daemonized and run in the background, it would poll the master in a configured interval to ask if a new or changed catalog is available. If that is the case, the catalog will get executed on the client.

Now,let'swriteasimplenodedefinition.Onyourlocalmachine,goto/etc/puppet/manifests and create a folder for nodes. Then edit the file /etc/puppet/manifests/site.pp and add this line:

```
import 'nodes/*'
```

This tells Puppet to import all files in the nodes folder, and this is where we will put our node definition. Create a file `nodes/node01.pp` with the content given in Listing 9-8:

Listing 9-8. *Creating a Node, with Puppet*

```
node 'node01.example.com' {
   file { "/tmp/test.txt":
    ensure => file,
    content => "It works!",
   }
}
```

If you rerun Puppet on the agent host, you will see that the Puppet master delivers a catalog to the client that is executed there. In this simple case, we will see the file `test.txt` after the execution.

After going through the features and benefits of Vagrant and Puppet, we are now ready to discover how a real-world project, the Jenkins CI build server, runs those tools in its DevOps approach.

Use Case: Open Source Infrastructure with Puppet

Have you ever downloaded a `.jar` file from some ASF (Apache Software Foundation) project, downloaded a Debian `.iso` from `kernel.org`, or maybe just read an article on Wikipedia? Have you ever stopped to think about how those bits get from one place to you? Who runs those machines, sets up those networks, receives those pages when all of that breaks?

Functional infrastructure is an oft overlooked but necessary component of the thriving open source ecosystem. In the examples above, there are probably a few people whose part- or full-time job it is to care for and tend to those systems.

Those are all big, important projects though, so what about the rest of us?

The Need, from the Jenkins Viewpoint

For a project such as Jenkins (`http://jenkins-ci.org`), the infrastructure needs had grown large enough so it no longer fit on a "forge" or could comfortably be hosted on some community member's Linode instance. Jenkins has multiple machines in multiple locations, such as the OSUOSL (`http://osuosl.org`) and even in Amazon's EC2.

Initially Jenkins had done things the old-fashioned way, that is to say the wrong way: manually handcrafting machine after machine and tweaking configurations until things appeared to work on machines in production. This continued until a (very) costly misconfiguration caused a single machine to inadvertently serve many terabytes of data more than it was supposed to, an expensive mistake Jenkins only noticed and corrected after someone received an overage bill.

After this "event," Jenkins started to incorporate Puppet into their management toolkit to meet a few crucial goals. The infrastructure should be:

- Testable (outside of production of course)
- Auditable
- Transparent

Unfortunately, migrating existing, handcrafted, infrastructure over to be managed by Puppet has been a long and tedious process. Imagine rebuilding a jet engine, midflight, while trying to maintain a steady altitude; that's hard.

The How at Jenkins

From the beginning, the Jenkins Puppet work has been married to Vagrant. Vagrant integrates neatly with Puppet and allows a user to effortlessly bring up a Linux-based virtual machine and provision it with Puppet. Let's go into more details about the different aspects.

Running Puppet

Puppet can be run in one of two ways: in a client-server architecture and with "stand-alone mode," the former being the far more common use case. The client-server model has its benefits, such as being able to push changes immediately, or gather so-called facts from all your nodes, but for Jenkins a stand-alone mode is used.

Jenkins wanted to introduce a way to remove additional service dependencies; being in multiple locations means you have one of two options with the client-server model: introduce a single point of failure in one location, or deploy a Puppet master to each location.

There is also a question of simplicity; since Vagrant integrates with Puppet by utilizing this stand-alone mode, Jenkins effectively tests their manifests locally in a manner much more consistent with the production environment.

The Tradeoffs

As you might expect, giving up the client-server model means Jenkins had to duplicate some functionality themselves. For example, since Jenkins cannot push updates to machines, Jenkins has a module for autoupdating each machine from Git. This module, shown in Listing 9-9, manages the periodical pulling of updates from the Git repository and running each machine's root[16] manifest:

Listing 9-9. *Managing Git Pull Requests and Running Root Manifest*

```
class autoupdate {
    include autoupdate::setup
    Class["autoupdate::setup"] -> Class["autoupdate"]

    cron {
        "pull puppet updates" :
            command     => "(cd /root/infra-puppet && sh run.sh)",
            user        => root,
```

[16] According to the current solution how Jenkins hosts its own production system at `http://jenkins-ci.org`, the root user is used. Often it is considered to be a good practice to not use the root user and create and use a dedicated user instead.

```
            minute      => 15,
            ensure      => present;

        # Might as well clean these up at some point
        "clean up old puppet logs" :
            command     => "rm -f /root/infra-puppet/puppet.*.log",
            user        => root,
            hour        => 4,
            minute      => 30,
            weekday     => '*',
            ensure      => present;
    } }

class autoupdate::setup {
    exec {
        "setup_git_repo" :
            cwd     => "/root",
            creates => "/root/infra-puppet",
            command => "git clone git://github.com/jenkinsci/infra-puppet.git",
            require => Package["git-core"],
            # In the case of a new machine, we probably already have this
            unless  => "test -d /root/infra-puppet/.git",
            path    => ["/usr/bin", "/usr/local/bin"],
    }
}
```

Also because Jenkins is not using the client-server module, this means Jenkins can't take advantage of some of the recent developments in Puppet module dependency management (example usage: `http://puppetlabs.com/blog/using-puppet-modules-to-install-and-manage-wordpress`). The approach taken has been to use Git submodules to manage the dependency on other modules to be reused, such as:

- `puppetlabs-stdlib`: common dependency for many of Puppet labs' modules
- `puppetlabs-firewall`: for managing `iptables` rules
- `puppetlabs-ntp`: for automatically setting up `ntp`, `crons`, and so forth
- `puppet-sshd`: for managing `sshd\_configs` and keeping the daemons running
- `puppet-concat`: for managing or concatenating configurations for services that do not support `conf.d/` style directories
- `puppet-postgres`: setting up `PostgreSQL` permissions, databases, and so forth

With puppet-module-tool,[17] Puppet can create, install, and search for modules on the Puppet Forge (see http://forge.puppetlabs.com), the place to find and share Puppet modules. The puppet-module-tool has been moved into core as of Puppet version 2.7.12.

Most Puppet modules are published to Puppet Forge, but their source typically lives on Github, making them easy to add as submodules, for example:

```
> cd project-root
> git submodule add git://github.com/puppetlabs/puppetlabs-firewall.git
modules/firewall
> git commit -m "Add the firewall module to manage iptables"
```

After it's been added to the tree, using it on a particular machine is easy enough (note: the firewall module adds some resources, some modules may require an "include" to use properly), as shown in Listing 9-10:

Listing 9-10. *Extending Our Configuration by Adding Firewall Settings*

```
firewall {
   '000 accept all icmp requests' :
       proto  => 'icmp',
       action => 'accept';

   '001 accept inbound ssh requests' :
       proto  => 'tcp',
       port   => 22,
       action => 'accept';

   # etc, etc
}
```

The submodule approach can be a double-edged sword and will cut you if you're not careful. On the one hand, you've got some level of built-in version freezing in adding the submodule. But if the submodule is updated too frequently or is under constant iteration, then you will litter the superproject with commits just to update the submodule reference.

Regardless of whether the client-server or stand-alone approach is used, reusing existing modules from the Puppet Forge is a good decision and can help bootstrap a project quickly, allowing you to focus on your specific needs instead of reinventing the wheel.

Source of Truth

One other major difference in this stand-alone setup compared to the client-server model is the "source of truth" for the machines managed by Puppet. In the traditional client-server architecture, the Puppet master is the boss, and what the boss says, goes. In the Jenkins setup, GitHub is our sole source of truth. This means that whatever the state of the "master" branch in our repository, that is the state of the machines (give or take 15 minutes).

The second major benefit of this approach is that Jenkins can easily accept contributions to the infrastructure by way of GitHub pull requests. You could submit a pull request (theoretically)

[17] https://github.com/puppetlabs/puppet-module-tool.

to add an entire new service on the Jenkins infrastructure, and it could be reviewed and merged without the "core" team ever giving you access to the hardware itself.

Testing with Vagrant

A lot of the flexibility Jenkins has derived from our Puppet setup hasn't come solely from Puppet, but the combination of Puppet and Vagrant. It's hard to imagine a more convenient way to set up a full BIND9 nameserver[18] in less than 40 minutes.

Jenkins needed a nameserver, and it needs a nameserver quickly. Fortunately, it already had an existing Puppet infrastructure set up, so it wasn't too much work to add the nameserver.

First things first; Jenkins created a branch to do the work in and then set up some folders (some paths changed to protect the innocent):

```
> git checkout -b build-a-nameserver-quick
> mkdir -p modules/bind/manifests modules/bind/files
> vim modules/bind/manifests/init.pp
```

After about 10 minutes of looking up package names, they had a basic set of resources defined to make up a simple BIND9-based nameserver, as shown in Listing 9-11:

Listing 9-11. *Adding BIND9 to Our Configuration*

```
# Skip the class declaration for now
package {
    'bind' :
        ensure => present, # Ensure the package is installed on Ubuntu
        name   => 'bind9';
}
service {
    "bind" :
        ensure  => running, # Ensure we have it up and running on boot
        require => Package['bind'],
        name    => 'bind9';
}
# Open up our firewall (provided by puppetlabs-firewall) to allow
# both TCP and UDP-based DNS traffic
firewall {
   "900 accept tcp DNS queries" :
       proto  => "tcp",
       port   => 53,
       action => "accept";
```

[18] BIND stands for Berkeley Internet Name Domain and is an implementation of the domain name system (DNS) protocol. More about BIND and its version 9 can be found at `http://www.bind9.net`.

```
    "901 accept udp DNS queries" :
        proto  => "udp",
        port   => 53,
        action => "accept";
}
```

If you've ever set up BIND before, you know that getting things installed and running isn't the hard part. It's the confusing, fickle configuration files that are really difficult to get right. Using Vagrant, we would enter a tight bind-frustration-and-iteration loop:

```
> vagrant up # also provisions when the VM is online
> vagrant ssh # Log into our VM
# Check to make sure that everything looks right, oh wait, no it doesn't,
# why are you doing that BIND? Why! Argh! You're so stupid! Don't act like
# that or I'll shut you down, that's it!
> vagrant destroy # I'll show BIND who's boss!
> vim init.pp # Okay, where'd we go wrong.
```

In your case, you might not use the `vagrant destroy` command as much as Jenkins does, but it's hard to go too long without wiping the slate clean (which the `'destroy'` command does) and starting all over again with just your pure Puppet manifests.

Around minute 35 of the "build a nameserver quick" project, Jenkins had things up and running and was running `dig(1)` from a local machine against the nameserver running inside the Vagrant VM, making final checks that everything was fully functional before sending code to GitHub:

```
> git commit modules/bind -m "Long and descriptive commit message should go here"
> git push origin
```

Within 15 minutes, the appropriate machine comes online and is a fully tested, code-reviewed, reproducible, and functional nameserver.

Full Circle

Unfortunately, the goal of the Jenkins project is not to build a textbook infrastructure, but rather to build one that works and stays working, so Jenkins can focus on building a stellar CI server (which Puppet Labs uses, we might add here: `http://jenkins.puppetlabs.com`). We can go back every day and figure out how and why things are the way they are, and it's all right there in the Git repository.

Thanks to Puppet, with a dash of Vagrant, a lot of things "just work" without much hand-holding, and that's how Jenkins is able to have a big infrastructure without a big-time investment.

Where to Look Next?

Puppet can be used together with Augeas[19] to enable you to alter existing files without the need to use templates.

[19] `http://augeas.net`.

If you start using Puppet with a number of hosts, you might want to look closer at the Puppet Dashboard, a web application that lets you monitor the state of all your Puppet-managed nodes.

You may want to take a look at PuppetDB,[20] which is a Puppet data warehouse to manage, storage, and retrieve all platform-generated data.

There are also many existing Puppet modules created by the community that you can find in the Puppet Forge, a repository of reusable Puppet modules.

All of the mentioned tools offer extensive documentation on their web pages and have either mailing lists, Google groups, Internet relay chat channels, or even all of them. Take a look at their web sites for more information.

If you are looking for further literature on Puppet, there are two recent books available: *Pro Puppet* by James Turnbull and Jeffrey McCune (Apress, 2011) and the *Puppet 2.7 Cookbook* by John Arundel (Packt Publishing, 2011). The former gives a good overall introduction to Puppet, also covering advanced topics and a lot of best practices. The latter contains a lot of small and easy-to-reuse recipes to solve your everyday Puppet problems.

Alternatives

With Vagrant and Puppet, we've examined two tools that can help to aid your development, testing, and operations. The choice of these, however, was based solely on my personal preferences, and there are many other configuration management frameworks out there, with the other two big open source tools being CFEngine[21] and Chef.

CFEngine is the oldest of these open source tools, with initial efforts dating back as early as 1993. The initial releases of Puppet and Chef are from 2005 and 2009, respectively. Due to its age, CFEngine is probably most widespread, however, in recent years, both Chef and Puppet gained a strong community and are used by a lot of startups and younger companies. There are also a lot of tools evolving from these two, addressing topics like integrated drive electronics or test-driven development, which may make these two more worthy of consideration. Just take some time to look into them and see which one you like better.

There are also commercial and a lot of lesser known frameworks and tools available. Check out Wikipedia or your usual development forum for more hints.

Conclusion

We've only scratched the surface of the possibilities provided by the tools addressed in this chapter. There is a lot more to be discovered, and I hope this has stirred your curiosity about using these or similar tools.

With configuration management and the infrastructure as code paradigm in general, the collaboration of development and operations is no longer merely a possibility, but almost a necessary. After all, every developer needs to know his or her share of system administration, and every system administrator has to learn about the components the architects and developers choose to run their application. The tools described here can greatly improve the sharing of knowledge between the two groups and help to make the DevOps movement even more successful.

Let's now proceed to the last chapter of this book that discusses acceptance tests.

[20] docs.puppetlabs.com/puppetdb.
[21] `http://cfengine.com`.