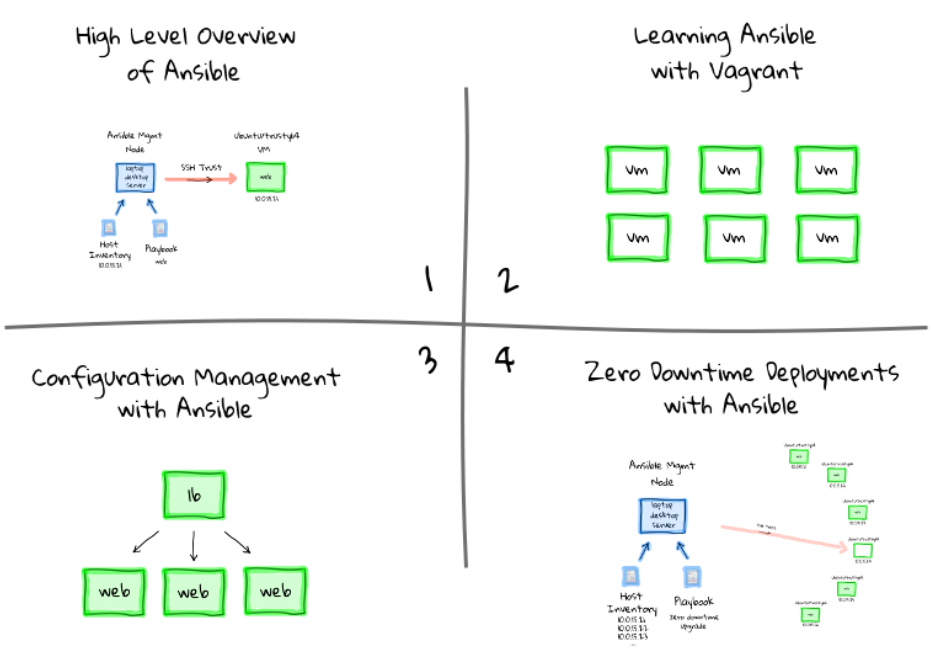
1.) We are also going to look at configuration management in general, and how this is an improvement over doing things manually, in the hopes that this will give Ansible some context.

2.)In part two, we are going to setup a bunch of virtual machines using Vagrant. We will work through installing Ansible from scratch, how it operates at the command line, what the configuration files look like, and how communication works between nodes.

3.)part three, we will look at using Ansible for configuration management tasks, by taking generic virtual machines from our Vagrant environment, and turning them into a web cluster, using haproxy and some nginx web servers.

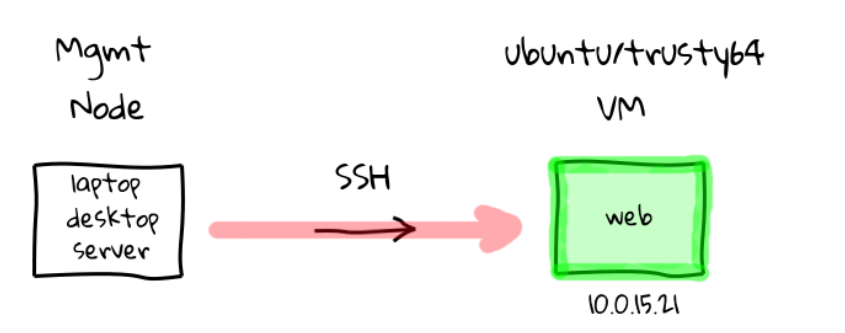
4.)Finally, in part four, we will use Ansible to do a zero-downtime rolling software deployment across a cluster of web nodes. The idea is that we can deploy code across a fleet of machines, without any downtime by using Ansible to orchestrate the various tasks for us, this will come in really handy for continues deployment workflows.

* [Ansible is Simple IT Automation](http://www.ansible.com/home)
* Learning Ansible with Vagrant
* Configuration Management with Ansible
* Zero-downtime Deployments with Ansible



**Before Configuration Management**

I thought it might make sense to give you a brief overview of my take on the problem Ansible is trying to solve.



Left-(Management Node) Right(web-server)

Imagine for a minute that you have two machine. The one of the left could be your laptop, desktop, or even a server you have access too, we will call it the management node.

The one on the right is a fresh Ubuntu virtual machine that we have just installed and booted, in reality this could be bare metal physical box, or even a cloud instance. **Let’s say we wanted to turn this freshly installed Ubuntu machine into a web server, say for example we are running Apache hosting a website, a rails app, or something similar**.

How would we go about doing that?

Well, we could do this:

1.) Manually by sshing into the machine, using its IP address 10.0.15.21, and

2.) Running the commands to install your application stack,

3.) Editing the configuration files by hand, and

4.) Finally copying over our application code.

5.) Once you are all done, you disconnect, and the machine is configured a working.

This is common practice, but this is also pretty manual work, and even if we have the steps documented somewhere, each machine is generally its own little snowflake depending on who installed it.

(**4.***Slang* A person who is considered to be overly sensitive or too easily offended, especially as a result of believinghimself or herself to be unique or special.)

-🡪This manual work can quickly compound if you need to do this across tens, or hundreds of machines.

-🡪It is also a real pain when one of the machines dies, because we are not really sure how it was created, or how we go about recreating it quickly. Do not get me wrong, I have manually installed many hundreds of machines, so I know this issue well, and it is extremely unpleasant when something important dies, and you are not really sure how to recreate it quickly.

(In that, you have a general idea of what was on there, but you engage in the same type of manual work that created this mess in the first place, potentially wasting many hours of your time on drop work type tasks. Constantly installing packages, dependencies, checking to see if it works, installing more packages, and tweaking things manually until the service is restored. It turns into this vicious cycle, and we are recreating the exact scenario that caused this mess in the first place.)

#### Enter Ansible

There has to be a better way right? Well, this is where configuration management tools come in. At a basic level, they are tools designed to automate away much of this manual work. Saving you time, reducing stress, and generally improving the process of creating machines in a timely manner. So, lets chat about Ansible.

Ansible is a free and open-source tool, mainly use on UNIX-like machines, which is directly targeted at solving this type of manual work problem.

**Lets start our example over again, and have a look at how we would solve this problem using Ansible.**

We start with our two machines again, the one of the left could be our laptop, desktop, or even a server, but this time call it the Ansible management node, because this is where we will install the Ansible software. The one on the right, is our freshly install Ubuntu machine, and this can be anywhere from our local machine running a VM, a physical box, or a cloud instance.

Once Ansible is installed on the management node, you will typically need two configuration files, something called **a host inventory, and the other a playbook**. The host inventory, is basically just a listing of hostnames, or IP addresses, for machines that we want to manage, and how they should be group together. In the example here, our Ubuntu box has the address 10.0.15.21, so we would just add this address to our host inventory, likely under a web group, since we want its end state to be a web server.

#### Hosts Inventory

you can group them together under arbitrary headings too. You use these brackets to create a group, then put the group name inside.

in this case we have a load balances group, and it has three hosts assigned to it. For hosts you want to manage with Ansible, you can use their fully qualified hostnames names, short hostnames, or IP addresses. You can see down here in the webservers group, we added our Ubuntu virtual machines IP address from the diagrams. You can also call these groups anything you want, lets change the name from webserver, to just web.

**[loadbalancer]**

haproxy-01.example.com

haproxy-02

10.0.15.15

**[web]**

web1

web2

10.0.15.21

**[testing]**

demo.example.com

#### Playbooks

Playbooks are configuration files that outline tasks that should be performed against hosts in the hosts inventory.

In our example today, we are setting up a web server on our freshly installed Ubuntu box, so we would likely want to install apache, or nginx, update the web server configuration files, deploy our application content, restart the server, etc.

Basically, all of the tasks we would have done manually, just documented in a configuration file called a playbook, so that Ansbile knows what to do.

Playbooks allow you to define each of these steps, in a simple, and quick to understand format, that pretty much anyone not familiar with Ansible can read and understand.

🡪 Lets quickly have a look at an example web server playbook.

we have defined our hosts group which we want to run this playbook against, this should look familiar from our hosts inventory.

We have this sudo line here, this allows Ansible to connect as a non-root user, then elevate to root for tasks like installing packages, adding users, or deploying configuration files.

Next we have a series of tasks that we want to run against each host in hosts inventory web group. You can see here that we want to install nginx, update our custom nginx configuration file, deploy our website code via a git pull from version control, and finally make sure nginx is started.

Also…in our example today, we are going to be mainly looking at using playbooks, these are configure files that outlines tasks that should be run, but you can also run Ansible in an ad-hoc mode, basically We will look at this heavily in parts two, three, and four of this episode series, but this ad-hoc mode can be thought of as a smart parallel ssh engine, at least that is what I think of it as, just with a lot more cool functionality built in.

---

- hosts: web

sudo: yes

tasks:

- name: install nginx

apt: name=nginx state=installed update\_cache=yes

- name: write our nginx.conf

template: src=templates/nginx.conf.j2 dest=/etc/nginx/nginx.conf

notify: restart nginx

- name: deploy website content

git: repo=https://github.com/jweissig/episode-47.git

dest=/usr/share/nginx/html/

version=release-0.01

- name: start ntp

service: name=nginx state=started

handlers:

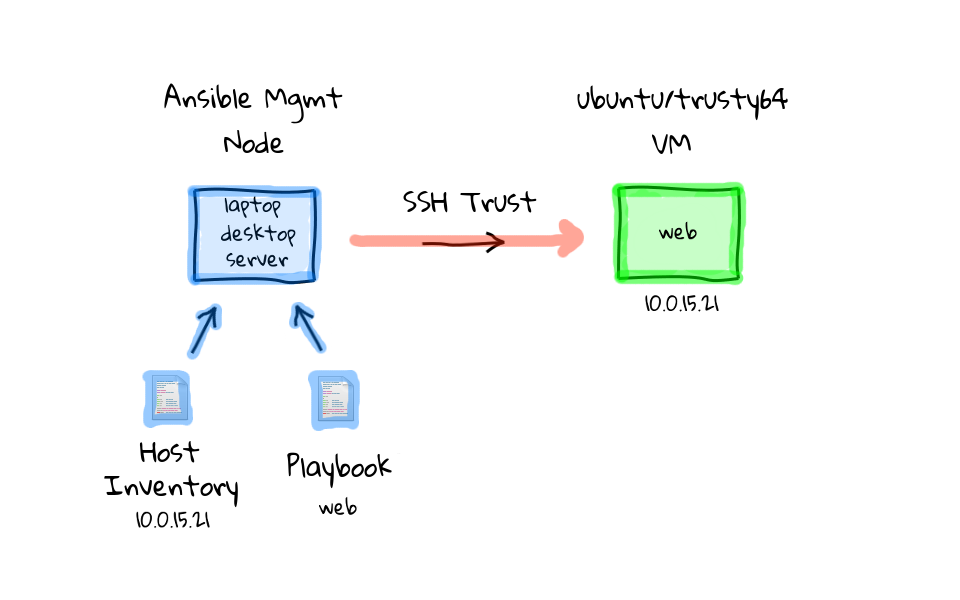
- name: restart nginx

service: name=nginx state=restarted

#### With Configuration Management

Let’s, have a look at how we would turn our Ubuntu virtual machine into a web server using Ansible. You run Ansible Playbook on the management node, it looks through the playbook that you have defined as a command argument, and notices that we are targeting nodes in the web group.

Ansible then reads in the hosts inventory to find nodes assigned to the web group. At this point, Ansible is ready to get to work, so it will remotely connect via ssh to the defined machines, typically you will want to have some type of ssh trust established via pre-shared keys, so that you do not have to enter the password all the time. Ansible will then start to step through the playbook tasks, one task at a time, going through them sequentially, from top to bottom, just like you would have done if logging in manually. So, it installs the packages, updates the configuration files, deploys our website code by using git, and finally starts our web service. When Ansible is happy that everything worked as expected you will get a status report saying that everything is good



So, that is basically the default Ansible workflow in a nutshell. But, you will soon notice that Ansible is a pretty flexible tool, and there are exceptions to pretty much everything I have shown you so far. For example, the hosts inventory can be a database if you have thousands of hosts, you also do not always need playbooks, you can run ad-hoc commands too. You can swap out ssh for a queue type system, if you find that performance is an issue for your number of hosts. But, for these examples today, I thought I would just show you what I think is the default mode of operation, when first playing around with Ansible.

#### How is Ansible is different?---Chef and puppet

1.) Probably the most glaring difference is that Ansible pushes the configuration out to each managed machine via ssh.

2.)Ansible only requires that you install the Ansible software onto a management node, and that the remote machines is running ssh with python installed, which every major distribution come with by default, there are no remote agents that you need to install, and everything is done via ssh from the management node. This make getting going with Ansible very easy, and upgrades are a snap, because you only need to update the Ansible install on one machine.

3.)You will often hear to term, batteries included, when reading about Ansible, that is because there are over 250 helper modules, or functions included with Ansible.

These allow you to construct playbooks to smartly add users, install ssh keys, tweak permissions, deploy code, install packages, interact with cloud providers for things like launch instances or modifying a load balancer, etc.

4.) I should mention that, even though Ansible is using ssh to connect to these remote machines, your playbooks and ad-hoc commands will almost always be using these 250 plus modules to smartly do things**. I guess what I am trying to say, is that Ansible is not simply running remote commands like a shell script would do, for automating package installs, adding users, etc. There is tons of logic built into these modules, and I encourage you to check these manual pages, just to get a sense of what can be done.**

5.)Ansible is also great for both ops and dev, because you do not have to give out root. Ansible is just using ssh to log into the remote machines, so there is clear separation of duties if needed, based on what account Ansible is using.

6.) One last pro for Ansible before we move on, there is a published Ansible Best Practices guide, and it is absolutely fantastic in the advice it offers. Things like **how to layout your files, using version control, naming, how to keep things simple.**

#### Taking it a Step Further

#### C:\Users\vebalusu\Desktop\43-ansible-multi-node-deployment-workflow.png

Once you have your playbooks figured out, you can use them as a starting point to launch any number of new machines, or turn existing machines, into something that you want.

Lets say for example, that instead of launch just a single new machine, maybe you need a bunch of extra capacity in your web tier, let’s say for a holiday rush. So, we launch a bunch of new generic virtual machines, and to manage these new machines with Ansible, we just add them to our hosts inventory like we did before, except this time we add all six machines.

This might be where you want to use a database, or some type of cloud module to poll the new instances that you created, so that you do not have to do this manually. Once all of these addresses are added into the hosts inventory, lets re-use our web playbook. Then lets follow the same process as before, run ansible playbook on our management node, that pulls in our web playbook, and that playbook references our updated hosts inventory. You can probably guess what is going to happen next. Again, Ansible connects to each of these machines in parallel through an ssh trust, from there it starts to run our defined tasks from the playbook, things like installing a web server, deploying our configuration file, starting the service, etc. Finally, each of these tasks completes, we have our installed and ready to use machines, along with a summary from Ansible of how everything went.

So, we can just reuse our previous playbook, where we deployed one machine, and use that same thing to deploy an entire set of new machines. The limit of how many machines you can talk to via ssh, is really based off your management nodes resources, things like bandwidth, and cpu.

**There are real world use cases where Ansible manages many thousands of nodes, people like Twitter, and the Fedora project both use Ansible. If you are looking for a little more details about how Twitter uses Ansible, check out this Youtube video, and I always find it interesting to see how other poeple are using these tools.** There is also a pretty decent slide deck about how the **Fedora project is using Ansible too, and the Fedora project actually publishes their hosts inventory, and playbooks, which are absolutely amazing resources for seeing how larger projects are implementing Ansible at scale**.

#### Zero-downtime Rolling Deployments

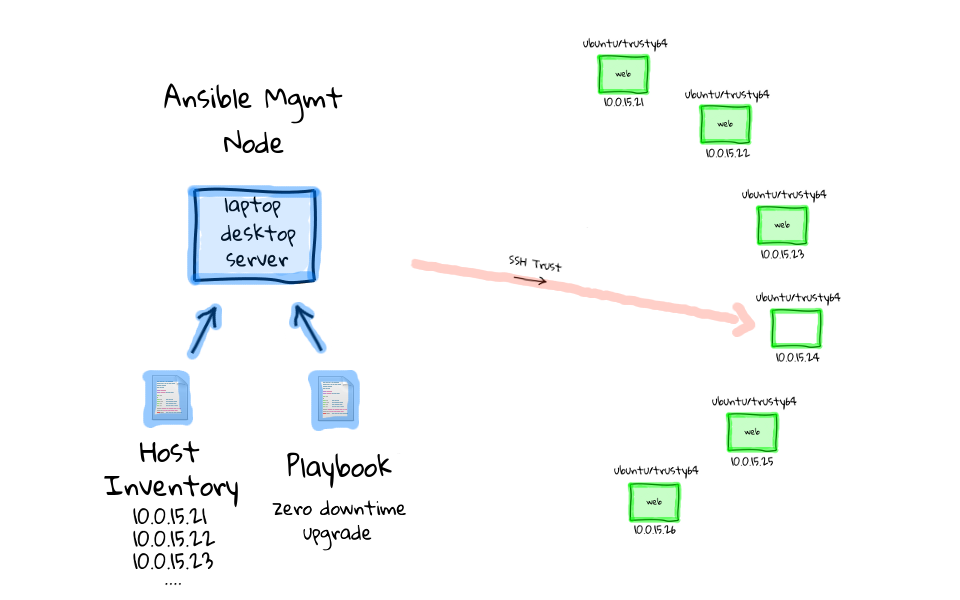
Up until this point we have used something called a web playbook, but lets switch gears for a minute, and look at using a different playbook, remember that these are just configuration files that you can easily edit, or create entirely new once, since they are just files. So, lets change that from a web playbook to something called the **zero downtime upgrade playbook, we** will actually cover this in part four of our episode series. The idea for this example is that this is how many people are implementing continues deployments across their infrastructure.

You will typically have a load balancer in front of all these web boxes, then you will use some type of tool, in our case Ansible, to notify the load balancer that this machine should be taken out of the pool, then you will update the software, and add it to the pool again. I will show you a live demo of this in part four, along with all the code, so that you can do this on your own**. It is pretty neat that we can use Ansible to orchestrate this entire sequence of events for us.**

I should mention that Ansible has this concept of pre and post task handlers, so we will define our tasks as deploying new software to a machine,

but the pre task will be to notify the load balancer that we want to remove this node from server, then we will update the software on our node, and finally the post task handler will add it back into the pool.

Lets just look at how this would work across our cluster of machines here. So we run Ansible Playbook again, this time it pulls in our zero-downtime upgrade playbook, then we pull in our hosts inventory, again you could use some type of database if you have a very dynamic environment. Finally, we use our ssh trust based on a preshared key, to connect to each box, one after another, removing it from the load balancer, updating the software, re-adding it to the load balancer, then moving onto the next node.



I have actually done these types of tasks manually before, and it really sucks, because there are so many moving parts, and errors are really easy to make. This is a major limiting factor to releasing software updates out into production, because it takes a long time, and you are fearful of breaking something. I actually found a really good article which talks about this, and I thought I would end the episode on it.

**The Guardian released a blog posting recently about how they went from 25 manual deployments of their site per year, to over 24,000 highly automated deployments, using an in house build continues deployment tool.** This is pretty much exactly the use case that I just talked about. Although, The Guardian is not using Ansible, their struggle really rings a bell for what many people are going though. If you have the time, I highly suggest reading their story, as I think it illustrates much of the thinking, rational, and work that does into deploying something like this.

Okay, so that pretty much wraps up with episode, and hopefully I have wet your appetite for the types of problems that Ansible is able to solve. The remaining episode parts should be out shortly, sometime in the next week, so check back soon.

#### About Episode - Duration: 46 minutes, Published: 2015-03-19

In this episode, we are going to play around with Ansible via four Vagrant virtual machines. We will install Ansible from scratch, troubleshoot ssh connectivity issues, review configuration files, and try our hand at common commands.

In this episode, we are going to get hands on with Ansible, by look at patterns for solving common Sysadmin tasks. Then, in parts three and four, we are going to take it to the next level, by deploying a web cluster, and doing a zero-downtime rolling software deployment.

#### Prerequisites

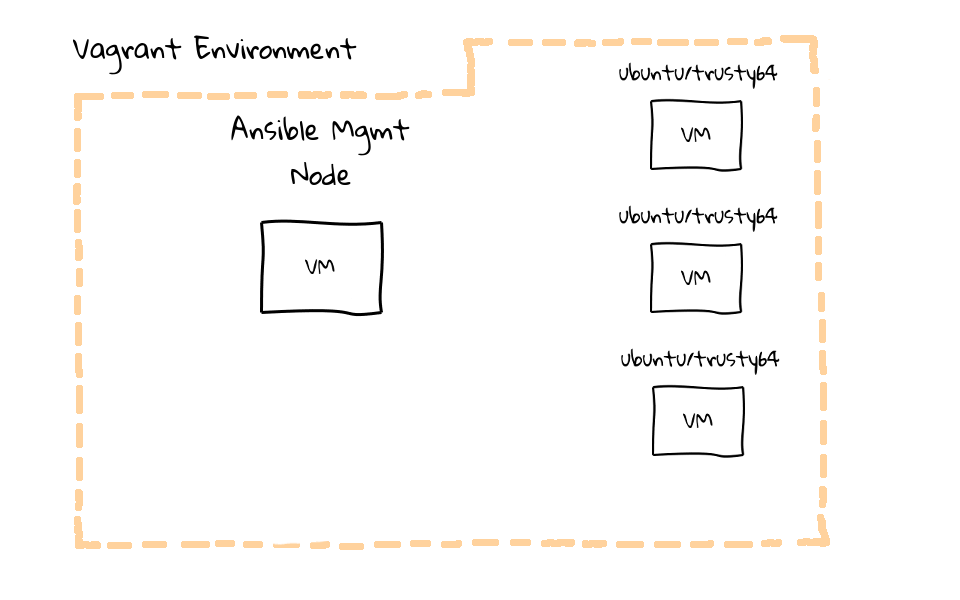
Crash Course on Vagrant

#### Vagrant Environment

Ansible being installed on a type of management machine, this will typically be your desktop, laptop, or some type of well connected server. Then from there, you use Ansible to push configuration changes out, via ssh.

Well, in this episode, we are actually going to create a Vagrant virtual machine, install Ansible on it, and have that act as the management node. But, we are also going to create three additional Vagrant virtual machines, and they will be our example client nodes managed by Ansible.

I should mention, Vagrant actually has an Ansible provisioning option, but it requires you to install Ansible on your local machines, outside of the Vagrant environment.

I chose to make an additional virtual machine inside our Vagrant environment, to act as the management node, rather than having you install software onto your system. The reason is that, the management node aspect of Ansible only works on UNIX like machines, so if you have Windows, you would not be able to try the demos. Also, I hate installing software on my local machine, just to try something out, and I really wanted to make sure this was a turn key solution for anyone wanting to play around with Ansible. 

**Supporting Materials**

[~]$ cd e45

[e45]$ ls -l

total 16

-rw-rw-r-- 1 jw jw 648 Feb 3 22:34 bootstrap-mgmt.sh

drwxrwxr-x 5 jw jw 4096 Feb 9 11:29 examples

-rw-rw-r-- 1 jw jw 867 Jan 30 21:48 README

-rw-rw-r-- 1 jw jw 1305 Feb 11 10:07 Vagrantfile

#### Launching Vagrant Environment

In the supporting examples archive, you will find three files, and an examples directory. Running vagrant status, and as you can see there are four virtual machines defined, a management node, a load balancer, and two web servers.

[e45]$ vagrant status

Current machine states:

mgmt not created (virtualbox)

lb not created (virtualbox)

web1 not created (virtualbox)

web2 not created (virtualbox)

These are all in the not created state, so lets fire them up by running, vagrant up. This will launch our four machines into the Vagrant environment, install Ansible on to the management node, and copy over our code snippets. I sped up the video a little here, this took about three and a half minutes to boot in real time.

[e45]$ vagrant up

Bringing machine 'mgmt' up with 'virtualbox' provider...

Bringing machine 'lb' up with 'virtualbox' provider...

Bringing machine 'web1' up with 'virtualbox' provider...

Bringing machine 'web2' up with 'virtualbox' provider...

==> mgmt: Importing base box 'ubuntu/trusty64'...

Now that we have launched the Ansible test environment, lets run vagrant status again, and you can see that everything is in a running state.

[e45]$ vagrant status

Current machine states:

mgmt running (virtualbox)

lb running (virtualbox)

web1 running (virtualbox)

web2 running (virtualbox)

The just launched Vagrant environment has four virtual machines, the Ansible management node, a load balancer, and two web servers.

I used a **Vagrant bootstrap post install script**, which I will show you in a minute, to install Ansible on the management node, configure a hosts inventory, along with moving example codes snippets over to be used in the demos today.

#### The README file

README file has an overview, providing a basic manifest, along with a link back to the episode series page.

The Vagrantfile defines what our multi-node test environment looks like. The examples directory, contains code snippets, which are used throughout the demos section of this episode series.

Vagrant allows you to boot virtual machines, then execute a shell script right after, using something called the shell provisioner. We used the shell bootstrap provisioner on the management node to install Ansible, copy over our example code snippets, and add some hosts entries to simplify networking within the environment.

bootstrap-mgmt.sh - Install/Config Ansible & Deploy Code Snippets

examples - Code Snippets

README - This file ;)

Vagrantfile - Defines Vagrant Environment

#### Vagrantfile

The Vagrantfile is where are four virtual machines are defined, and **you will notice three blocks of code here, one for our Ansible management node, one for a load balancer, and the final one is for our web servers.**

Vagrant.configure("2") do |config|

config.vm.box = "bento/ubuntu-16.04

config.vm.network "public\_network

config.vm.provider "virtualbox" do |vb|

vb.memory = "1024"

# Enable provisioning with a shell script. Additional provisioners such as

# Puppet, Chef, Ansible, Salt, and Docker are also available. Please see the

# documentation for more information about their specific syntax and use.

# config.vm.provision "shell", inline: <<-SHELL

# apt-get update

# apt-get install -y apache2

# SHELL

-🡪

# Defines our Vagrant environment

#

# -\*- mode: ruby -\*-

# vi: set ft=ruby :

Vagrant.configure("2") do |config|

# create mgmt node

config.vm.define :mgmt do |mgmt\_config|

mgmt\_config.vm.box = "ubuntu/trusty64"

mgmt\_config.vm.hostname = "mgmt"

mgmt\_config.vm.network :private\_network, ip: "10.0.15.10"

mgmt\_config.vm.provider "virtualbox" do |vb|

vb.memory = "256"

end

mgmt\_config.vm.provision :shell, path: "bootstrap-mgmt.sh"

end

# create load balancer

config.vm.define :lb do |lb\_config|

lb\_config.vm.box = "ubuntu/trusty64"

lb\_config.vm.hostname = "lb"

lb\_config.vm.network :private\_network, ip: "10.0.15.11"

lb\_config.vm.network "forwarded\_port", guest: 80, host: 8080

lb\_config.vm.provider "virtualbox" do |vb|

vb.memory = "256"

end

end

# create some web servers

**# https://docs.vagrantup.com/v2/vagrantfile/tips.html**

(1..2).each do |i|

config.vm.define "web#{i}" do |node|

node.vm.box = "ubuntu/trusty64"

node.vm.hostname = "web#{i}"

node.vm.network :private\_network, ip: "10.0.15.2#{i}"

**node.vm.network "forwarded\_port", guest: 80, host: "808#{i}"**

node.vm.provider "virtualbox" do |vb|

vb.memory = "256"

**end**

**end**

**end**

**end**

management node. We are telling Vagrant to define a new virtual machine, using the Ubuntu Trusty 64bit image, set the hostname to mgmt, configure a private network address, this will allow all of our virtual machines to communicate with each other over known addresses. Next, we set the virtual machine memory to 256 megs. This should work on most of your machines, as I tried to keep the resource limits down. **Finally, we tell Vagrant to run our bootstrap management node script, this downloads and install Ansible, deploys code snippets for our examples, and finally adds /etc/hosts file entries for machines in our vagrant environment.**

load balancer virtual machine. For this episode, we are just going to use this as a regular client node, not a load balancer just yet. In parts three and four of this episode series, we are going to configure this node as a haproxy load balancer using Ansible, and then setup a bunch of web nodes behind it. Should be a pretty cool Ansible example use case.

We do pretty much the same thing here, define a machine, set the image to use, configure the hostname, define a private network address, configure a port map so that we can connect to the load balancer, and set the memory limit. **The port map line, is likely the only interesting bit here, this maps a port from our machine running Vagrant, to the load balancer virtual machine.** This is really useful for testing network connected software, or in our case for parts three and four, testing a haproxy load balancer, with a bunch of web servers behind it.

The last block is almost exactly the same as the load balancer block, **except that we have this each statement here, which allows us to set the number of machines to be launched, via this number range. This is a bit of a trick, but it allows for a number of web nodes to launched, rather than having many duplicate code blocks for each web box.** You can read this Vagrant tips pages about it. In this episode, we are only going to launch two web servers, however in parts three and four, we will bump this number up as we play around with the load balancer.

#### Bootstrap Ansible Script

Now that you know how our Vagrant environment is configured, lets check out how Ansible gets installed on to the management node, along with the hosts inventory, and our example code snippets.

# create mgmt node

config.vm.define :mgmt **do** |mgmt\_config|

mgmt\_config.vm.box = "ubuntu/trusty64"

mgmt\_config.vm.hostname = "mgmt"

mgmt\_config.vm.network :private\_network, ip: "10.0.15.10"

mgmt\_config.vm.provider "virtualbox" **do** |vb|

vb.memory = "256"

**end**

mgmt\_config.vm.provision :shell, path: "bootstrap-mgmt.sh" # <---

**end**

Lets jump back to our editor and check out how this is triggered. When we launched our Vagrant environment via the Vagrantfile, our management node boots, and references the bootstrap script as a type of post install script.

#!/usr/bin/env bash

# install ansible (http://docs.ansible.com/intro\_installation.html)

apt-get -y install software-properties-common

apt-add-repository -y ppa:ansible/ansible

apt-get update

apt-get -y install ansible

# copy examples into /home/vagrant (from inside the mgmt node)

cp -a /vagrant/**examples/\* /home/vagrant**

**chown -R vagrant:vagrant /home/vagrant**

# configure hosts file for our internal network defined by Vagrantfile

cat >> /etc/hosts

# vagrant environment nodes

10.0.15.10 mgmt

10.0.15.11 lb

10.0.15.21 web1

10.0.15.22 web2

10.0.15.23 web3

10.0.15.24 web4

10.0.15.25 web5

10.0.15.26 web6

10.0.15.27 web7

10.0.15.28 web8

10.0.15.29 web9

The four commands from earlier, off the Ansible documentation site, are actually really simple. **First we install a supporting package, add the Ansible software repository, update the package cache, and finally install Ansible via apt-get install.**

Next, we copy over the code snippets that will be used for the demos, I am not going to cover these too much here, as we will go into detail in just a bit.

**Finally, we add host values to our management nodes /etc/hosts file, these correspond to the preset network addresses used in our Vagrantfile.** You can use IP addresses with Ansible too, but I like to use hostnames, as it makes things a little more personal, and easy to understand the machines you are talking to.

#### Connecting to the Management Node

Now that you have an idea of how this all fits together, lets head back to the command line, and test this Vagrant environment out. We can log into the management node, by running vagrant ssh mgmt

[e45]$ vagrant ssh mgmt

vagrant@mgmt:~$ uptime

vagrant@mgmt:~$ lsb\_release -a

Description: Ubuntu 14.04.1 LTS

vagrant@mgmt:~$ pwd

/home/vagrant

vagrant@mgmt:~$ ls -l

total 68

-rw-r--r-- 1 vagrant vagrant 50 Jan 24 18:29 ansible.cfg

-rw-r--r-- 1 vagrant vagrant 410 Feb 9 18:13 e45-ntp-install.yml

-rw-r--r-- 1 vagrant vagrant 111 Feb 9 18:13 e45-ntp-remove.yml

-rw-r--r-- 1 vagrant vagrant 471 Feb 9 18:15 e45-ntp-template.yml

-rw-r--r-- 1 vagrant vagrant 257 Feb 10 22:00 e45-ssh-addkey.yml

-rw-rw-r-- 1 vagrant vagrant 81 Feb 9 19:29 e46-role-common.yml

-rw-rw-r-- 1 vagrant vagrant 107 Feb 9 19:31 e46-role-lb.yml

-rw-rw-r-- 1 vagrant vagrant 184 Feb 9 19:26 e46-role-site.yml

-rw-rw-r-- 1 vagrant vagrant 133 Feb 9 19:30 e46-role-web.yml

-rw-rw-r-- 1 vagrant vagrant 1236 Feb 9 19:27 e46-site.yml

-rw-rw-r-- 1 vagrant vagrant 105 Feb 9 01:04 e47-parallel.yml

-rw-rw-r-- 1 vagrant vagrant 1985 Feb 10 21:59 e47-rolling.yml

-rw-rw-r-- 1 vagrant vagrant 117 Feb 9 01:03 e47-serial.yml

drwxrwxr-x 2 vagrant vagrant 4096 Jan 31 05:03 files

-rw-r--r-- 1 vagrant vagrant 67 Jan 24 18:29 inventory.ini

drwxrwxr-x 5 vagrant vagrant 4096 Feb 9 19:26 roles

drwxrwxr-x 2 vagrant vagrant 4096 Feb 6 20:07 templates

The files prefixed with e45, correspond the episode 45, so we will be looking at these in this episode. In episode 46, we will be looking at developing a web cluster, using haproxy, and nginx, these are the playbooks we will use. Then, in episode 47, we will look at doing multiple rolling website deployment across our cluster of nginx nodes, fronted by the haproxy load balancer.

**If we run df, you can see that we have the trademark /vagrant mount, this links back to our host machine, into our project directory. If we list the directory contents, you can find the supporting scripts used to launch this environment, just in case you need to copy something over.**

vagrant@mgmt:~$ df -h

Filesystem Size Used Avail Use% Mounted on

/dev/sda1 40G 1.2G 37G 3% /

none 4.0K 0 4.0K 0% /sys/fs/cgroup

udev 115M 12K 115M 1% /dev

tmpfs 24M 340K 24M 2% /run

none 5.0M 0 5.0M 0% /run/lock

none 120M 0 120M 0% /run/shm

none 100M 0 100M 0% /run/user

vagrant 235G 139G 96G 60% /vagrant

vagrant@mgmt:~$ ls -l /vagrant

total 16

-rw-rw-r-- 1 vagrant vagrant 676 Feb 12 08:12 bootstrap-mgmt.sh

drwxrwxr-x 1 vagrant vagrant 4096 Feb 9 19:29 examples

-rw-rw-r-- 1 vagrant vagrant 867 Jan 31 05:48 README

-rw-rw-r-- 1 vagrant vagrant 1305 Feb 11 18:07 Vagrantfile

#### Review Ansible Install

vagrant@mgmt:~$ ansible --version

ansible 1.8.2

configured module search path = None

When you first get started with Ansible, you will likely edit something called the ansible.cfg file, and the hosts inventory.ini file.

When you run Ansible, it will check for an ansible.cfg in the current working directory, the users home directory, or the master configuration file located in /etc/ansible. I have created one in our Vagrant users home directory, because we can use it to override default settings, for example, I am telling Ansible that our hosts inventory is located in /home/vagrant/inventory.ini.

Lets have a look and see what the inventory.ini file actually looks like. You might remember from part one of this series, that the **inventory.ini is made of up machines that you want to manage with Ansible.** A couple things are going on here, we have a lb group up here, this is for our load balancer box, next we have the web group, and it is made up of web1, and web2. So, these are our active web nodes, and then down here, we have our commented out ones, these will be added in parts three and four, of this episode series.

vagrant@mgmt:~$ cat ansible.cfg

[defaults]

hostfile = /home/vagrant/inventory.ini

vagrant@mgmt:~$ cat inventory.ini

[lb]

lb

[web]

web1

web2

#web3

#web4

#web5

#web6

#web7

#web8

#web9

**You might be wondering where these IP addresses and hostnames actually come from. Well, our Vagrantfile assigns a predetermined IP address to each box, then our bootstrap post install script adds these known entries to the /etc/hosts file, and that gives us name resolution for use in our hosts inventory.**

vagrant@mgmt:~$ cat /etc/hosts

127.0.0.1 localhost

127.0.1.1 mgmt mgmt

# vagrant environment nodes

10.0.15.10 mgmt

10.0.15.11 lb

10.0.15.21 web1

10.0.15.22 web2

10.0.15.23 web3

10.0.15.24 web4

10.0.15.25 web5

10.0.15.26 web6

10.0.15.27 web7

10.0.15.28 web8

10.0.15.29 web9

vagrant@mgmt:~$ ping lb

PING lb (10.0.15.11) 56(84) bytes of data.

64 bytes from lb (10.0.15.11): icmp\_seq=1 ttl=64 time=0.826 ms

64 bytes from lb (10.0.15.11): icmp\_seq=2 ttl=64 time=0.652 ms

--- lb ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 1002ms

rtt min/avg/max/mdev = 0.652/0.739/0.826/0.087 ms

vagrant@mgmt:~$ ping web1

PING web1 (10.0.15.21) 56(84) bytes of data.

64 bytes from web1 (10.0.15.21): icmp\_seq=1 ttl=64 time=0.654 ms

64 bytes from web1 (10.0.15.21): icmp\_seq=2 ttl=64 time=0.652 ms

--- web1 ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 1001ms

rtt min/avg/max/mdev = 0.652/0.653/0.654/0.001 ms

vagrant@mgmt:~$ ping web2

PING web2 (10.0.15.22) 56(84) bytes of data.

64 bytes from web2 (10.0.15.22): icmp\_seq=1 ttl=64 time=1.32 ms

64 bytes from web2 (10.0.15.22): icmp\_seq=2 ttl=64 time=1.74 ms

--- web2 ping statistics ---

2 packets transmitted, 2 received, 0% packet loss, time 1002ms

rtt min/avg/max/mdev = 1.321/1.533/1.745/0.212 ms

I just wanted to explain why we are using our own ansible.cfg and inventory.ini file, because this will hopefully round out our understanding of how to override defaults in the environment.

In this environment, we want Ansible to use our custom inventory file, so we need to modify the defaults. You can find the Ansible stock configuration files located in /etc/ansible. **Lets open up the ansible.cfg file for a second. It is actually pretty long and there are plenty of tweaks you can make, but the one thing I wanted to point out, was this comments section up top here. You will notice that, Ansible will read in the ansible.cfg in the current working directory, dot ansible.cfg in the users home directory, or the global configuration located in /etc/ansible, whichever it finds first. So, when we run ad-hoc Ansible commands, or playbooks, it knows to use the local ansible.cfg file, rather than the stock defaults.**

vagrant@mgmt:~$ cat ansible.cfg

[defaults]

hostfile = /home/vagrant/inventory.ini

vagrant@mgmt:~$ cd /etc/ansible/

vagrant@mgmt:/etc/ansible$ ls -l

total 12

-rw-r--r-- 1 root root 8156 Dec 4 23:11 ansible.cfg

-rw-r--r-- 1 root root 965 Dec 4 23:11 hosts

vagrant@mgmt:/etc/ansible$ less ansible.cfg

# config file for ansible -- http://ansible.com/

# ==============================================

# nearly all parameters can be overridden in ansible-playbook

# or with command line flags. ansible will read ANSIBLE\_CONFIG,

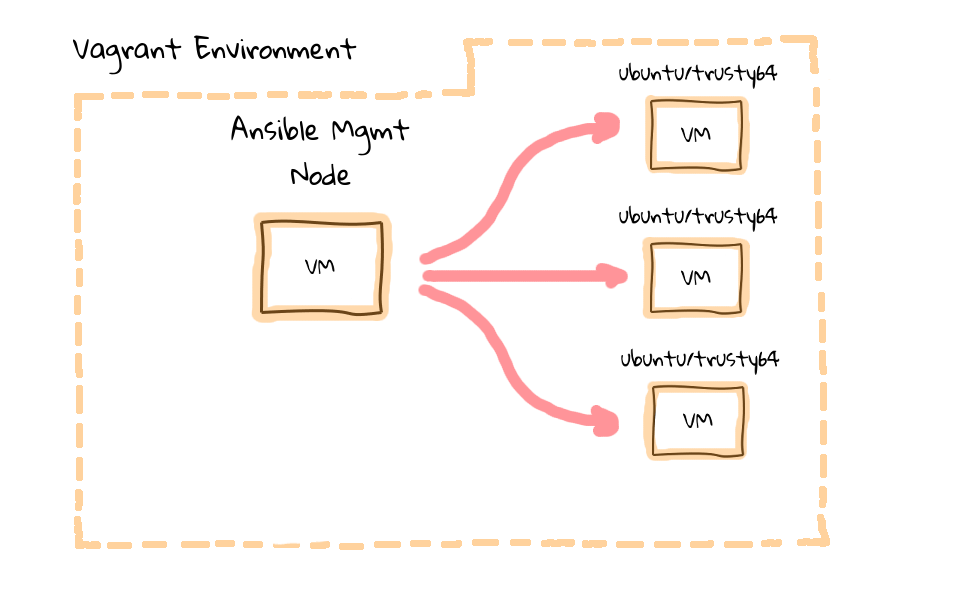
# ansible.cfg in the current working directory, .ansible.cfg in

# the home directory or /etc/ansible/ansible.cfg, whichever it

# finds first

#### SSH Connectivity Between Nodes

As we heavily discussed in part one of this episode series, Ansible connects out to remote machines via ssh, there are no remote agents.



One issue that we need to deal with, is when you have not connected to a machine before, you will be prompted to verify the ssh authenticity of the remote node. Lets test with out by sshing into web1.

vagrant@mgmt:~$ ssh web1

The authenticity of host 'web1 (10.0.15.21)' can't be established.

ECDSA key fingerprint is 28:4e:9c:ee:07:f8:6f:91:05:27:d4:b9:e3:03:4a:bf.

Are you sure you want to continue connecting (yes/no)? no

Host key verification failed.

You can see that we are prompted to verify the authenticity of web1. **Well, this error will cause Ansible to prompt you, and likely give you an error message. Let me show you what I am talking about, lets run ansible web1 -m ping**, this uses the ping module to verify we have connectivity with a remote host. As you can see, we are prompted, then given an error message, saying that we cannot connect**. I could accept this, but what happens if I wanted to connect to 25, 50, or a 100 machines for the first time? Do you want to constantly be prompted?**

vagrant@mgmt:~$ ansible web1 -m ping

The authenticity of host 'web1 (10.0.15.21)' can't be established.

ECDSA key fingerprint is 28:4e:9c:ee:07:f8:6f:91:05:27:d4:b9:e3:03:4a:bf.

Are you sure you want to continue connecting (yes/no)? no

web1 | FAILED => SSH encountered an unknown error during the connection. We recommend you re-run

the command using -vvvv, which will enable SSH debugging output to help diagnose the issue

There are several ways of dealing with this issue. **You could turn off the verification step via the ssh config file, what about manually accepting the prompts, or you can use ssh-keyscan to populate the known\_hosts file, with the machines from your environment.**

Lets run ssh-keyscan against web1, and you can see that web1, returns two ssh public keys, that we can use to populate our known\_hosts file. This will get around the authenticity issue, and is a quick way of accepting these keys manually, and we still have protection against man in the middle attacks going forward.

vagrant@mgmt:~$ ssh-keyscan web1

# web1 SSH-2.0-OpenSSH\_6.6.1p1 Ubuntu-2ubuntu2

web1 ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBHYXqCD5alepd.....

# web1 SSH-2.0-OpenSSH\_6.6.1p1 Ubuntu-2ubuntu2

web1 ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQCwmTC7j6/9+l05tsuEDWbRX5KtZ60vzw8jSicYKJuqmPStq.....

So lets run, ssh-keyscan lb, for our load balancer machine, web1, and web2, then we will pipe the output into our Vagrant users .ssh/known\_hosts file.

vagrant@mgmt:~$ **ssh-keyscan lb web1 web2 >> .ssh/known\_hosts**

# web2 SSH-2.0-OpenSSH\_6.6.1p1 Ubuntu-2ubuntu2

# lb SSH-2.0-OpenSSH\_6.6.1p1 Ubuntu-2ubuntu2

# lb SSH-2.0-OpenSSH\_6.6.1p1 Ubuntu-2ubuntu2

# web1 SSH-2.0-OpenSSH\_6.6.1p1 Ubuntu-2ubuntu2

# web1 SSH-2.0-OpenSSH\_6.6.1p1 Ubuntu-2ubuntu2

# web2 SSH-2.0-OpenSSH\_6.6.1p1 Ubuntu-2ubuntu2

I also like to verify that what I just did worked, so lets cat the known\_hosts file, and see if it looks okay. Cool, so we have just added a bunch of keys from our remote machines.

vagrant@mgmt:~$ cat .ssh/known\_hosts

#### Hello World Ansible Style

I think of this as the equivalent of a, Hello World program, for Ansible. Finally, since we are using ssh to connect remotely as the vagrant user, we want ansible to prompt us to enter the password

vagrant@mgmt:~$ ansible all -m ping --ask-pass

SSH password:

lb | success >> {

"changed": false,

"ping": "pong"

}

**I would like to demonstrate that connections are cached for a little while too.** We can actually run the same ping command again, but this time remove the ask pass option, you will notice that the command seemed to execute quicker, and that we were not prompted for a password. Why is that?

vagrant@mgmt:~$ ansible all -m ping

web1 | success >> {

"changed": false,

"ping": "pong"

Lets also have a look at the established network connections, between our management node, and client machines.

Keeping ssh connections around for a while, greatly speeds up sequential Ansible runs as you remove the overhead associated with constantly opening and closing connections to many remote machines.

vagrant@mgmt:~$ ps x

PID TTY STAT TIME COMMAND

3077 ? S 0:00 sshd: vagrant@pts/0

3078 pts/0 Ss 0:00 -bash

3443 ? Ss 0:00 ssh: /home/vagrant/.ansible/cp/ansible-ssh-web2-22-vagrant [**mux**]

3446 ? Ss 0:00 ssh: /home/vagrant/.ansible/cp/ansible-ssh-lb-22-vagrant [**mux**]

3449 ? Ss 0:00 ssh: /home/vagrant/.ansible/cp/ansible-ssh-web1-22-vagrant [mux]

3523 pts/0 R+ 0:00 ps x

vagrant@mgmt:~$ netstat -nap |grep EST

(Not all processes could be identified, non-owned process info

will not be shown, you would have to be root to see it all.)

tcp 0 0 10.0.2.15:22 10.0.2.2:48234 ESTABLISHED -

tcp 0 0 10.0.15.10:34471 10.0.15.21:22 ESTABLISHED 3449/ansible-ssh-we

tcp 0 0 10.0.15.10:52373 10.0.15.11:22 ESTABLISHED 3446/ansible-ssh-lb

tcp 0 0 10.0.15.10:37684 10.0.15.22:22 ESTABLISHED 3443/ansible-ssh-we

#### Establishing a SSH Trust

**A very common use-case, will be that you want to establish password less access out to your client nodes, say for a continuous integration system, where you are deploying automated application updates on a frequent basis**. A key piece of that puzzle, is to establish a SSH trust, between your management node and client nodes.

What better way to do that than using Ansible itself via a Playbook. Lets check out the e45-ssh-addkey.yml file. I am just going to cat the file contents here, so we can work through what this playbook does, then we can run it against our client nodes.

vagrant@mgmt:~$ cat e45-ssh-addkey.yml

---

- hosts: all

sudo: yes

gather\_facts: no

remote\_user: vagrant

tasks:

- name: install ssh key

authorized\_key: user=vagrant

key="{{ lookup('file', '/home/vagrant/.ssh/id\_rsa.pub') }}"

state=present

**This is what a pretty basic playbook will look like.**

We use this hosts option, to define which hosts in our hosts inventory, we want to target. Next, we say that we want to run these commands via sudo. Why is this important to have in here? Well, you typically connect via ssh as some non-root user, in our case the vagrant user, then you want to do system level things, so you need to sudo as root. If you do not define the remote user, it will assume you want to connect as the current user, so in our case we are already the vagrant user, and this is redundant, but I put this in here so that we could chat about it.

Next, you will see this gathering facts option. By default, facts are gathered when you execute a Playbook against remote hosts, these facts include things like hostname, network addresses, distribution, etc. These facts can be used to alter Playbook execution behaviour too, or be included in configuration files, things like that. It adds a little extra overhead, and we are not going to use them just yet, so I turned it off.

The tasks section, is where we define the tasks we want to run against the remote machines. In this Playbook, we only have one task, and that is to deploy an authorized ssh key onto a remote machine, which will allow us to connect without using a password.

Each task will have a name associated with it, and it is just an arbitrary title that you can name anything you want, next you define the module that you want to use. We are using the authorized\_key module here. It is a module, which will help us configure ssh password less logins on remote machines. I have linked to this modules documentation page in the episode notes below, and it provides detailed options for everything you can do with it. Lets just briefly walk through it though. We tell the authorized\_key module, that we want to add an authorized\_key to the remote vagrant user, we define where on the management node we should lookup the key file from, then we make sure it exists on the remote machine.

If we look on our management node, in the .ssh directory, you will see that we do not actually have this public RSA key yet.

vagrant@mgmt:~$ ls -l .ssh/

total 8

-rw------- 1 vagrant vagrant 466 Feb 12 20:46 authorized\_keys

-rw-rw-r-- 1 vagrant vagrant 2318 Feb 12 21:59 known\_hosts

So, lets go ahead and create one, by running, **ssh-keygen -t, this specifies the type of key we want to create, lets use RSA, there are other newer key types out there, but RSA should be the most compatible if there are older systems you want to manage. Then, -b, this tells the ssh-keygen how long of a key we want, lets enter 2048.** If you are trying to do this outside of our vagrant environment, be careful not to overwrite any of your existing RSA keys, as that might cause you grief.

vagrant@mgmt:~$ ssh-keygen -t rsa -b 2048

Generating public/private rsa key pair.

Enter file **in** which to save the key (/home/vagrant/.ssh/id\_rsa):

Enter passphrase (empty **for** no passphrase):

Enter same passphrase again:

Your identification has been saved **in** /home/vagrant/.ssh/id\_rsa.

Your public key has been saved **in** /home/vagrant/.ssh/id\_rsa.pub.

The key fingerprint is:

81:2a:29:31:c5:cb:02:bf:4b:6a:06:c3:71:4b:e8:54 vagrant@mgmt

The key's randomart image is:

+--[ RSA 2048]----+

| .. |

|...E . |

|+oo. . . |

|.\*++ . . |

|=.\*.o S |

|o+oo |

|.+ . |

|.o. |

|o |

+-----------------+

I suggest you use ssh-agent to manage your ssh key passwords for you, but this is a little outside of the scope for this episode. I plan to cover this in the very near future. For now, lets just hit enter a couple times, without using a password, and our keys have been generated. Lets just verify that they actually exist, by listing the contents of .ssh, and it looks our ssh RSA public and private keys have been created.

vagrant@mgmt:~$ ls -l .ssh/

total 16

-rw------- 1 vagrant vagrant 466 Feb 12 20:46 authorized\_keys

-rw------- 1 vagrant vagrant 1679 Feb 12 22:10 id\_rsa

-rw-r--r-- 1 vagrant vagrant 394 Feb 12 22:10 id\_rsa.pub

-rw-rw-r-- 1 vagrant vagrant 2318 Feb 12 21:59 known\_hosts

Let have a peak at our e45-ssh-addkey.yml playbook again.

vagrant@mgmt:~$ cat e45-ssh-addkey.yml

---

- hosts: all

sudo: yes

gather\_facts: no

remote\_user: vagrant

tasks:

- name: install ssh key

authorized\_key: user=vagrant

key="{{ lookup('file', '/home/vagrant/.ssh/id\_rsa.pub') }}"

state=present

So, now that we have fulfilled the requirements of generating a local RSA public key for the Vagrant user, lets deploy it to our remote client machines. We do this by running, ansible-playbook, and then the playbook name, in this case e45-ssh-addkey.yml. We also need to add the ask pass option, since we do not have password less login configured yet. We are prompted for the password, lets just enter that again, and rather quickly our vagrant users public key has been deploy to our client nodes, which will allow us to connect via ssh without a password going forward.

vagrant@mgmt:~$ ansible-playbook e45-ssh-addkey.yml --ask-pass

SSH password:

PLAY [all] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK: [install ssh key] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

changed: [web1]

changed: [web2]

changed: [lb]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

lb : ok=1 changed=1 unreachable=0 failed=0

web1 : ok=1 changed=1 unreachable=0 failed=0

web2 : ok=1 changed=1 unreachable=0 failed=0

Great, so it looks like our ssh key deployment worked. Lets review what happened here. The task that we defined inside the Playbook, to install our management nodes ssh key, says that it changed three nodes, web1, web2, and our load balancer. You will notice that the name here, matches what we put in the playbook. Then in our playbook, we asked to deploy the authorized key from the management node, out to the remote client nodes, then down here, you can see it was actually changed on each of these remote machines.

Finally, we are given a playbook recap. The recap section provides you with the okay, changed, unreachable, and failed tallies per node.

Lets just rerun the ansible-playbook e45-ssh-addkey command, and remove the ask pass option, as we are not going to need that anymore. You will notice that in our previous playbook run, things changed, but in the most recent run, the tasks are all green, and the playbook recap is green too, meaning that nothing was changed.

vagrant@mgmt:~$ ansible-playbook e45-ssh-addkey.yml

PLAY [all] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

TASK: [install ssh key] \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

ok: [web2]

ok: [lb]

ok: [web1]

PLAY RECAP \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

lb : ok=1 changed=0 unreachable=0 failed=0

web1 : ok=1 changed=0 unreachable=0 failed=0

web2 : ok=1 changed=0 unreachable=0 failed=0

#### Ansible is Idempotent

You might hear the word idempotent, or idempotence, when working with configuration management software. What you are looking at here is a good example of it, and Wikipedia has a pretty good overview of what this means, basically that Ansible scripts can be run many times in a row, and nothing will change, unless a change needs to be made. So, in this example, Ansible is smart enough to check the clients authorized key file, and see that it already exists, so it does not do anything.

#### Verify SSH Trust

To ensure we were not piggy backing on a previously opened ssh session, and that our ssh trust is indeed working as we expect, lets run ps x again, and you can see that there are no active ssh sessions into our clients node. We can also verify there are no established ssh sessions, between the management node, and our client machines.

Lets run, ansible all -m ping, so we are running an ad-hoc ansible command, targeting all nodes, saying that we want to use a module, and then specifying the ping module. Looks like we are in good shape as everything worked as expected and without a password.

vagrant@mgmt:~$ ps x

PID TTY STAT TIME COMMAND

3077 ? S 0:00 sshd: vagrant@pts/0

3078 pts/0 Ss 0:00 -bash

3784 pts/0 R+ 0:00 ps x

vagrant@mgmt:~$ netstat -nap|grep EST

(No info could be read **for** "-p": geteuid()=1000 but you should be root.)

tcp 0 0 10.0.2.15:22 10.0.2.2:48234 ESTABLISHED -

vagrant@mgmt:~$ ansible all -m ping

lb | success >> {

"changed": false,

"ping": "pong"

}

web1 | success >> {

"changed": false,

"ping": "pong"

}

web2 | success >> {

"changed": false,

"ping": "pong"

}

So, at this point, we have all of these bits implemented. Our management node is up and running, with a valid Ansible install, we have password less access into our client nodes, and we can run commands on the client nodes. I know this is a bit verbose on my part, but these are all issues that you will likely want to resolve on your own, so I thought it would be worth the time to work through these issues together.

#### Package, File, Service Pattern

You might be wondering, what every day tasks is Ansible actually useful for? Good question, almost everyone who comes into contact with Ansible, will at one point or another, typically want to install packages, push out configuration files, and start, stop, or restart, remote services. So, lets focus our effort on these types of thing for the remainder of the episode.

#### Installing a Package

Next, we use the -a option, which allows us to pass an argument to the module we have selected.

vagrant@mgmt:~$ ansible web1 -m apt -a "name=ntp state=installed"

#### Deploying a Configuration File

Now that you know how to install packages, lets deploy a configuration file, for the ntp service. Included with the supporting material that you can download, is a files directory, and I have included a ntp.conf configuration file. I am just going to type the ad-hoc command, then we can work through what is going on. You should start to recognize commonalities with other commands we have run before.

vagrant@mgmt:~$ ls -l files/

total 8

-rw-r--r-- 1 vagrant vagrant 504 Jan 24 18:29 ntp.conf

-rw-r--r-- 1 vagrant vagrant 417 Jan 24 18:29 ntp.conf.j2

ansible web1 -m copy -a src=/home/vagrant/files/ntp.conf dest=/etc/ntp.conf mode=644 owner=root group=root –sudo. So, we are running an ad-hoc ansible command, targeting web1, saying that we want to use a module, and then specifying the copy module. The copy module allows you to, you guessed it, copy files from the management node, out to the client nodes. So, we are saying, copy this source file, which we looked at up here, set the destination to /etc/ntp.conf, then set the mode, owner, and group. Finally, since we are connecting as the vagrant user, we need to sudo to root, since this is a file owned by root.

vagrant@mgmt:~$ ansible web1 -m copy -a "src=/home/vagrant/files/ntp.conf dest=/etc/ntp.conf mode=644 owner=root group=root" --sudo

#### Stopping, Starting, and Restarting Services

vagrant@mgmt:~$ ansible web1 -m service -a "name=ntp state=restarted"

**Ah-hoc Shell Commands**

vagrant@mgmt:~$ ansible all -m shell -a "uptime"

web1 | success | rc=0 >>

We are returned the command output from all machines in our hosts inventory, you could limit this to various groups, or individual machines too.

vagrant@mgmt:~$ ansible all -m shell -a "uname -a"

power off all of the machines in a particular cluster. Obviously this is extreme, but you get the idea.

vagrant@mgmt:~$ ansible web -m shell -a "/sbin/reboot"

#### Playbooks

My personal take on Playbooks, is that they give you a way to bundle these ah-hoc commands, and associated options like sudo, into a configuration file. Kind of like, running commands at the terminal versus using a shell script. Same concept.

#### The Package, File, Service Pattern Playbook

If we list the directory contents on our management node again, and will see four playbooks for this episode, and you already know what the e45-ssh-addkey one does, but lets checkout the e45-ntp-install playbook.

vagrant@mgmt:~$ ls -l

...

-rw-r--r-- 1 vagrant vagrant 410 Feb 9 18:13 e45-ntp-install.yml

-rw-r--r-- 1 vagrant vagrant 111 Feb 9 18:13 e45-ntp-remove.yml

-rw-r--r-- 1 vagrant vagrant 471 Feb 9 18:15 e45-ntp-template.yml

-rw-r--r-- 1 vagrant vagrant 257 Feb 10 22:00 e45-ssh-addkey.yml

...

Lets just cat the file, then we can talk through what it does, and since the ad-hoc command modules are reused, you will likely already know what most of this stuff does.

vagrant@mgmt:~$ cat e45-ntp-install.yml

---

- hosts: all

sudo: yes

gather\_facts: no

tasks:

- name: install ntp

apt: name=ntp state=installed update\_cache=yes

- name: write our ntp.conf

copy: src=/home/vagrant/files/ntp.conf dest=/etc/ntp.conf mode=644 owner=root group=root

notify: restart ntp

- name: start ntp

service: name=ntp state=started

handlers:

- name: restart ntp

service: name=ntp state=restarted

First, we specify that we want to target all hosts, just like at the command line, where we were targeting web1. Next, we say that, yes, we want to sudo, this is because we are installing packages, deploying configuration files into places like /etc, and restarting services.

**What is cool about this, is that there can be a separation between root, and regular users that want to use Ansible.** For example, in many shared hosting environments, you will not have root access. But if you are not deploying your applications into places that require root, you can still happily use automation tools like Ansible, and this is a little different than other configuration management tools because we are only using ssh. **Not that this cannot be done with other tools, but I would put forward, that this is not something commonly done by default in other tools.**

**EPISODE-3**

### **Episode #46 - Configuration Management with Ansible.**

**In this episode, we are going create a load balanced web cluster using Vagrant and Ansible. The goal is to show how Ansible can solve real problems by building up complex infrastructure from scratch.**

#### Series Recap

Before we dive in, lets quickly review what this episode series is about. In part one, episode #43, we looked at what Ansible is at a high level, basically a comparison of doing things manually, versus using configuration management. Part two, episode #45, served as a bit of an Ansible crash course, we used both ad-hoc commands and playbooks, to deploy packages, configuration files, and restart services. In this episode, we are going to pick up where we left off in part two of this series, building out our Vagrant environment even more, to support a load balanced web cluster. That being said, you should have at least watched part two, before jumping into this one. Finally, in part four, we are going to close out this episode series, with a zero-downtime rolling website deployment across our web cluster nodes.

#### What We Are Building

The end result of our activities today, will be a Vagrant environment with a fully functioning haproxy load balancer, with six nginx web servers sitting behind it. We will use Ansible to install all of the required packages, deploy configuration files, and start the correct services on to each of these boxes. All without logging into any of these nodes manually.