Episode #45 - Learning Ansible with Vagrant (Part 2/4)

About Episode - Duration: 46 minutes, Published: Mar 19, 2015

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.

Download: mp4 (https://s3.amazonaws.com/sysadmincasts.com/static/videos/45-learning-ansible-with-vagrant-part-2-4.mp4) or webm (https://s3.amazonaws.com/sysadmincasts.com/static/videos/45-learning-ansible-with-vagrant-part-2-4.webm)

Get notified about future content via the mailing list (/get-notified), follow @jweissig_ (https://twitter.com/jweissig_) on Twitter for episode updates, or use the RSS feed (/feed.rss).

Links, Code, and Transcript

- GitHub: jweissig/episode-45 (Supporting Material) (https://github.com/jweissig/episode-45)
- Episode Playbooks and Examples (e45-supporting-material.tar.gz) (https://d1cg27r99kkbpq.cloudfront.net/static/extra/e45-supporting-material.tar.gz)
- Episode Playbooks and Examples (e45-supporting-material.zip) (https://d1cg27r99kkbpq.cloudfront.net/static/extra/e45-supporting-material.zip)
- Ansible is Simple IT Automation (http://www.ansible.com/home)
- Episode #43 - 19 Minutes With Ansible (Part 1/4) (/episodes/43-19-minutes-with-ansible-part-1-4)
- Episode #46 - Configuration Management with Ansible (Part 3/4) (/episodes/46-configuration-management-with-ansible-part-3-4)
- Episode #47 - Zero-downtime Deployments with Ansible (Part 4/4) (/episodes/47-zero-downtime-deployments-with-ansible-part-4-4)
- Episode #42 - Crash Course on Vagrant (revised) (/episodes/42-crash-course-on-vagrant-revised)
- Vagrant Documentation - Ansible Provisioning (https://docs.vagrantup.com/v2/provisioning/ansible.html)
- Vagrant Documentation - Vagrant Shell Provisioner (https://docs.vagrantup.com/v2/provisioning/shell.html)
- Vagrant Documentation - Forwarded Ports (http://docs.vagrantup.com/v2/networking/forwarded_ports.html)
- Vagrant Documentation - Tips & Tricks (https://docs.vagrantup.com/v2/vagrantfile/tips.html)
- Ansible - Installation (http://docs.ansible.com/intro_installation.html)
- Ansible - Configuration File (http://docs.ansible.com/intro_configuration.html)

- Ansible - Best Practices (http://docs.ansible.com/playbooks_best_practices.html)
- Ansible - Ad-Hoc Commands (http://docs.ansible.com/intro_adhoc.html)
- Ansible - Module Index (http://docs.ansible.com/modules_by_category.html)
- Ansible - Ping Module (http://docs.ansible.com/ping_module.html)
- Ansible - Playbooks (http://docs.ansible.com/playbooks.html)
- Ansible - Authorized_Key Module (http://docs.ansible.com/authorized_key_module.html)
- Wikipedia: Idempotence (http://en.wikipedia.org/wiki/Idempotence)
- Ansible - Apt Module (http://docs.ansible.com/apt_module.html)
- Ansible - Copy Module (http://docs.ansible.com/copy_module.html)
- Ansible - Service Module (http://docs.ansible.com/service_module.html)
- Ansible - Shell Module (http://docs.ansible.com/shell_module.html)
- Ansible - Handlers: Running Operations On Change (http://docs.ansible.com/playbooks_intro.html#handlers-running-operations-on-change)
- Canada - ca.pool.ntp.org (http://www.pool.ntp.org/zone/ca)
- Ansible - Setup Module (http://docs.ansible.com/setup_module.html)
- Ansible - Variables (http://docs.ansible.com/playbooks_variables.html)
- Ansible - Template Module (http://docs.ansible.com/template_module.html)

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.

Series Recap

Before we dive in, I thought it might make sense to 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. 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.

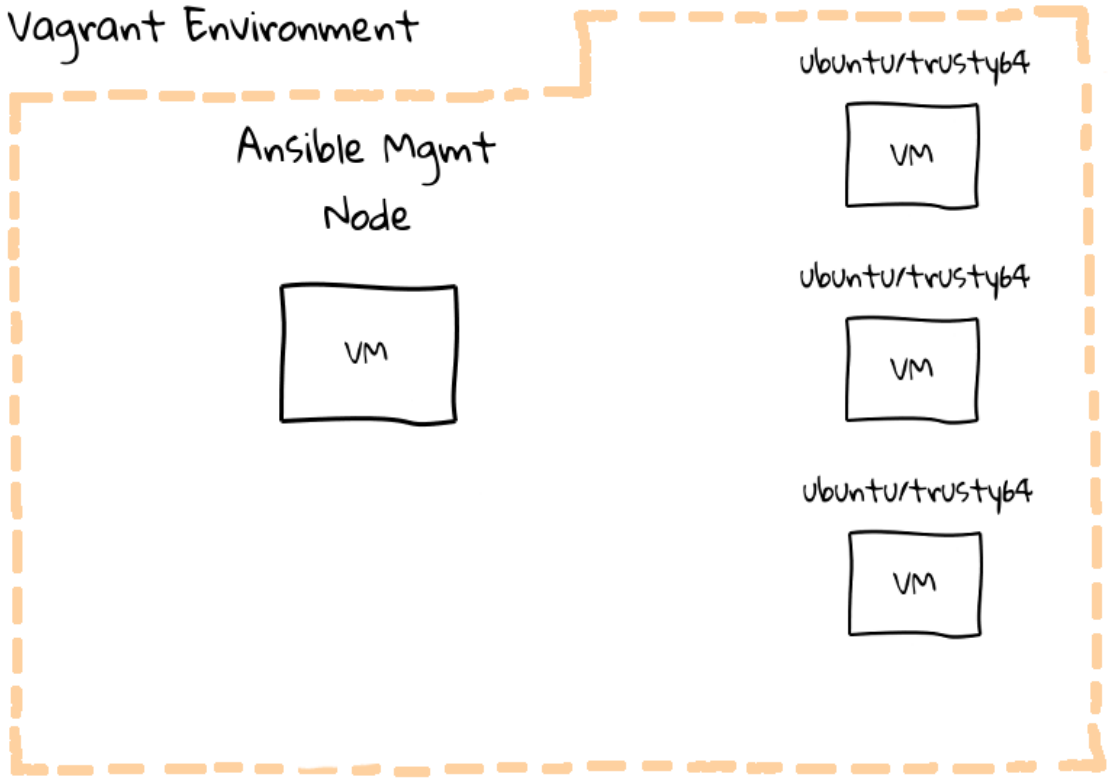
Prerequisites

If you have not already watched episode #42, where I gave a Crash Course on Vagrant, then I highly suggest doing that now. I am going to assume you know what Vagrant is, how it works, and that you have a working setup on your box. The reason that you need Vagrant, is that we are going to build a multi-node Vagrant testing environment, where we can play around with Ansible to get a feeling for how it works. That being said, you do not actually need to download these demos today, but I have put together a number of supporting examples, so that if you wanted to play around with Ansible on your own, there should be nothing stopping you.

Vagrant Environment

Let me show you what we are going to be building today by way of a couple diagrams. Hopefully you will remember, back in part one of this series, I tried to drive the point home about 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.



Our Vagrant environment is going to look like this. With the management node created as a virtual machine, sitting alongside our client test nodes. There are a few benefits to this setup too, in that our Vagrant environment used to learn Ansible, will be cross platform, totally isolated, and you do not need to install anything on your local machines. We are totally free to test destructive changes.

Supporting Materials

I have bundled the Vagrantfile, and all supporting demo files, into a compressed archive which can be downloaded via the links below this video. You can download and extract it into a new directory, which you can also use as your Vagrant project directory. I have already done that on my machine, and put them into a directory called, e45.

```
[~]$ 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. We will review these files in just a moment, first lets check the Vagrant environment, as defined in our Vagrantfile, by 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'...
==> mgmt: Matching MAC address for NAT networking...
==> mgmt: Checking if box 'ubuntu/trusty64' is up to date...
==> mgmt: Setting the name of the VM: e45_mgmt_1423697801980_68834
....
==> web2: Machine booted and ready!
==> web2: Checking for guest additions in VM...
==> web2: Setting hostname...
==> web2: Configuring and enabling network interfaces...
==> web2: Mounting shared folders...
    web2: /vagrant => /home/jw/e45
```

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. This gives us a turn key learning environment, in a matter of a few minutes, and my hope is that you will find it really easy to use. The motivation for all this, is that we can quickly stand up an Ansible testing environment, where we can execute ad-hoc commands, and eventually playbooks, against our client nodes to get a feeling for how Ansible works in real life.

The README file

Now that you have seen how easy it is to launch this Vagrant environment, lets check out how it works under the hood, by opening up an editor, and reviewing what these supporting files do. The 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.

```
                        _                 _                       _
    Welcome to          | |         (_)                     | |
 ___ _   _ ___  __ _  __| |_ __ ___  _ _ __     ___ __ _ ___| |_ ___
/ __| | | / __|/ _` |/ _` | '_ ` _ \| | '_ \   / __/ _` / __| __/ __|
\__ \ |_| \__ \ (_| | (_| | | | | | | | | | | (_| (_| \__ \ |_\__ \
|___/\__, |___/\__,_|\__,_|_| |_| |_|_|_| |_|  \___\__,_|___/\__|___/
      __/ |
     |___/   #43, #45, #46, #47 - Learning Ansible with Vagrant

http://sysadmincasts.com/episodes/43-19-minutes-with-ansible-part-1-4


      File Manifest

  bootstrap-mgmt.sh - Install/Config Ansible & Deploy Code Snippets
           examples - Code Snippets
             README - This file ;)
        Vagrantfile - Defines Vagrant Environment
```

I just wanted to quickly review the Vagrantfile, and bootstrap script, so that you have an understanding of how this all fits together. This might also come in handy next time you need to create a test environment, say for one of your projects, as this is a pretty common pattern which can be reused.

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.

```
# 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
```

I thought it might make sense to, work through what each of these blocks does, starting with the 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.

Next, we configure a 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.

Hopefully, at this point I have explained how we get the four node environment configured. This is a pretty generic Vagrant pattern that I use for all types of tests, so I really wanted to show you how it works, in the hopes that you will find other uses for it too.

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.

The Ansible documentation site has some great instructions for getting you up and running. I just wanted to cover this, before I show you the management node bootstrap script, so that you have a fairly good idea of how this is all working. There are many different ways to install Ansible, these include installing from source, using operating system package managers, tools like apt, or yum, and then there are brew, and pip installer methods too. I always prefer to use the operating system package managers when possible, as it just makes my life easier, not having to maintain multiple package management systems. Ansible is very proactive about making sure these various repositories have the latest version, so you are not at a disadvantage using apt, or yum, over the source code version on Github. I have copied these four commands here, since we are using Ubuntu, and used them in our management node bootstrap script.

```
  # 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. The idea being, you put commands in here that will help configure this vanilla Ubuntu Trusty 64bit box, into something a little more useful.

```
#!/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. I thought it might make sense to just show you around the management node quickly, by running commands like uptime, getting the release version, just so that you can get a lay of the land. We are logged in as the vagrant user, and sitting in its home directory, if we list the directory contents here, you can see a wide variety of files.

```
[e45]$ vagrant ssh mgmt

vagrant@mgmt:~$ uptime
 20:57:19 up 11 min,  1 user,  load average: 0.00, 0.04, 0.05

vagrant@mgmt:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 14.04.1 LTS
Release:    14.04
Codename:   trusty

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

Now that we have a basic lay of the land, lets check out how Ansible is installed on this machine, and what some of the configuration files are that you will need to know about. For our first interactions with Ansible, we are going to just be looking at the ad-hoc Ansible command, then we will look at automating some things, through the use of Ansible Playbooks. You can check the version that you have, by running, ansible --version. Depending on when you run through this tutorial, you might have an updated version from what you see here, since the bootstrap script will download and install the latest Ansible version available.

```
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

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
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
```

We can verify connectivity, by pinging some of these machines. Lets ping the load balancer node, yup it seems to be up. What about web1, and web2. Great, they are both up. This is one thing that I love about Vagrant, is that we have configured a pretty complex environment, with just a couple scripts, and it is easy for me, to send it to you.

```
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
```

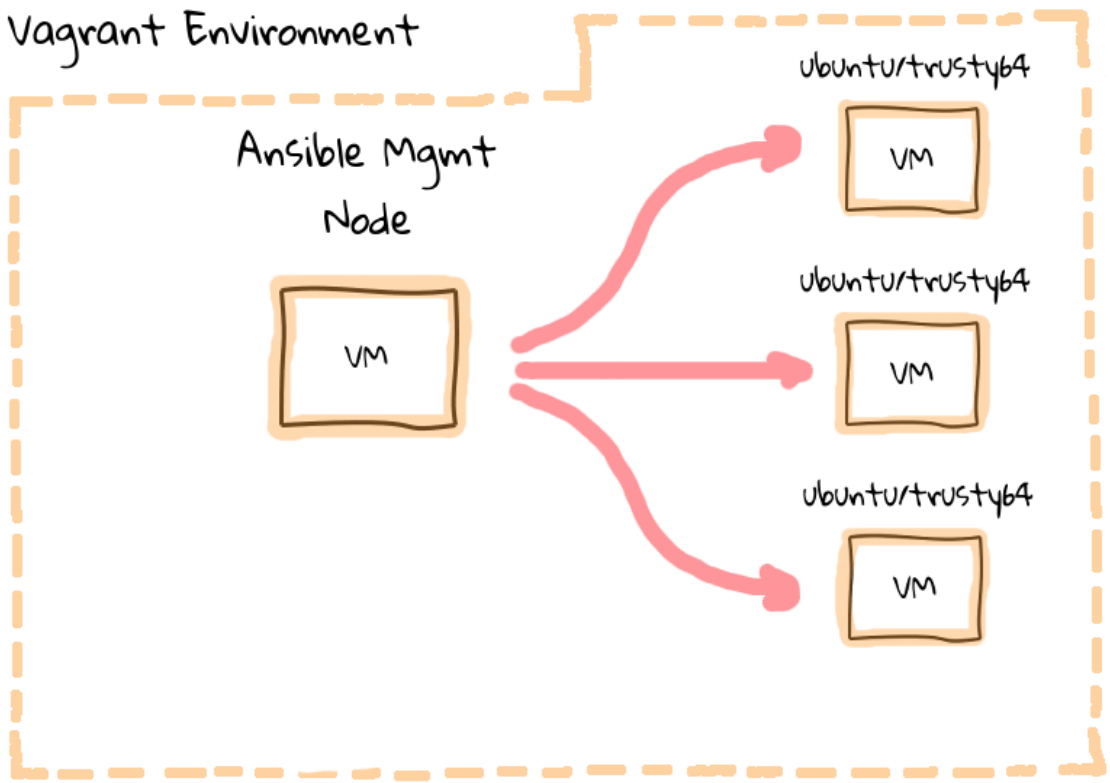
I just wanted to explain why we are using our own ansible.cfg and inventory.ini file, because this will hopefully round out our understanding of how to override defaults in the environment. In this environment, we want Ansible to use our custom inventory file, so we need to modify the defaults. You can 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
```

You can also check out the Ansible configuration manual page, and it has a break down of all the various tweaks you can make. There is also the Ansible best practices guide, which talks about using version control too, and there are tons of other tips in there. I highly suggest checking these pages out as it is just packed with useful guidance.

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? Granted, you might not run into this issue in your environment, as you likely already have verified the authenticity of these remote machines. But, I wanted to discuss how to deal with this, just in case it comes up.

```
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

At this point, I think we are finally ready to dive head first into the Ansible demos. We actually already ran an ad-hoc Ansible command, when testing ssh connectivity with web1, but lets have a look at how that worked. The hosts inventory plays a key role in making connections happen, we basically already covered this, but any nodes that you want to manage with Ansible, you put into this file, just like our load balancer, and our web nodes.

In this episode, we are going to explore a couple different ways to running Ansible. First, we will explore using ad-hoc Ansible commands, these are kind of one off tasks that you will run against client nodes. Then there are Ansible Playbooks, which allow you to bundle many tasks into a configuration file, and run this against client nodes. We will check Playbooks out in the latter part of this episode.

You can run ad-hoc commands, by typing ansible, then the first argument will be the host group, or machine, that you want to target. So, if we said ansible all, that would target all of the machines in our host inventory file. We could say, ansible lb, and that would target machines in the lb group, or ansible web, to target all of our web servers, but you can also target individual nodes, by using their hostnames, like web1, for example.

```
vagrant@mgmt:~$ cat inventory.ini
[lb]
lb

[web]
web1
web2
#web3
#web4
#web5
#web6
#web7
#web8
#web9
```

In this example, lets run ansible all, -m, this means that we want to use a module, there is actually a huge list of modules that you can choose from. Actually, lets have a peak at the documentation site for a minute. I actually mentioned this in part one, but it is really important to how Ansible functions. You will often hear to term, batteries included, when reading about Ansible, that is because there are over 250 plus helper modules, or functions included with Ansible. These allow you to construct ad-hoc commands, or 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. Each module has a dedicated page on the Ansible documentation site, along with detailed examples, and I have found this a major bonus to working with Ansible, in that things just work out of the box.

So, in this example, we are using the ping module, which is not exactly like a traditional ICMP ping, but rather it verifies that Ansible can login to the remote machine, and everything is working as expected. 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. The password is just vagrant, and you can see it in the top right hand corner of your video, just thought it might make sense to overlay it, then lets hit enter.

```
vagrant@mgmt:~$ ansible all -m ping --ask-pass
SSH password:
lb | success >> {
    "changed": false,
    "ping": "pong"
}

web1 | success >> {
    "changed": false,
    "ping": "pong"
}

web2 | success >> {
    "changed": false,
    "ping": "pong"
}
```

Great, we get back this successful ping pong message from each of our remote machines, and this verifies that a whole chain of things is working correctly. Our remote machines are up, ansible is working, the hosts inventory is configured correctly, ssh is working with our vagrant user accounts, and that we can remotely execute commands.

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? Well, lets have a look at the running processes for our Vagrant user, and as you can see we have three processes, one for each of our remote machines.

```
vagrant@mgmt:~$ ansible all -m ping
web1 | success >> {
    "changed": false,
    "ping": "pong"
}

lb | success >> {
    "changed": false,
    "ping": "pong"
}

web2 | success >> {
    "changed": false,
    "ping": "pong"
}
```

Lets also have a look at the established network connections, between our management node, and client machines. You can see that we have three open connections, and they link back to the process ids, we looked at earlier. If we turn name resolution on, you can see that we have one from the management node to web1, one to our load balancer, and finally, one to web2. 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

vagrant@mgmt:~$ netstat -ap |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 mgmt:ssh                10.0.2.2:48234          ESTABLISHED -
tcp        0      0 mgmt:34471              web1:ssh                ESTABLISHED 3449/ansible-ssh-we
tcp        0      0 mgmt:52373              lb:ssh                  ESTABLISHED 3446/ansible-ssh-lb
tcp        0      0 mgmt:37684              web2:ssh                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. You can call them anything you want, but they will typically have, dot yml extension. There is actually this great manual page over on the Ansible documentation site that goes into a crazy amount of detail. I think I have mentioned this before, but the documentation for the Ansible project, is some of the best I have seen for configuration management tools. Very clear and easy to understand. The one cool thing about Playbooks in general, is that they are basically configuration files, so you do not need to know any programming to use them.

Lets step through this playbook and see what is going on. 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

Say for example, that we want to install the ntp package onto web1. Maybe the clock is drifting, and we want to make sure it has accurate time keeping. So, lets run, ansible web1 -m apt -a name=ntp state=installed. Some of this likely already looks familiar, we are running an ad-hoc ansible command, targeting web1, saying that we want to use a module, and then specifying the apt module. The apt module allows you to install packages, just like you would on any Ubuntu machine, using the apt-get command. There is actually a really great module documentation page that you can checkout with all types of examples. We are looking at the apt package module here, but this would be functionally the same using the yum module, if you were on a CentOS machine. I have included links to both the documentation pages in the episode notes below.

Next, we use the -a option, which allows us to pass an argument to the module we have selected. So, we are saying, make sure the package with a name of ntp, is in the installed state. You will also notice, that we are doing this as the vagrant user, as we have not said to sudo. Meaning that, what we are about to run is going to fail, since the vagrant user does not have permission to install packages without sudoing to root. I just wanted to show you what an error message would look like.

After running the ad-hoc command, we get back a message in all red, and this is a pretty good indication that something went wrong without actually reading anything. We can see that apt-get install ntp failed. There is a permission denied error. Then there is even a note asking if we are root.

```
vagrant@mgmt:~$ ansible web1 -m apt -a "name=ntp state=installed"
web1 | FAILED >> {
    "failed": true,
    "msg": "'apt-get install 'ntp'' failed: E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission de
    to lock the administration directory (/var/lib/dpkg/), are you root?\n",
    "stderr": "E: Could not open lock file /var/lib/dpkg/lock - open (13: Permission denied)\nE: Unable to lock the
    directory (/var/lib/dpkg/), are you root?\n",
    "stdout": ""
}
```

Well, we connected as the Vagrant user, and tried to install a package, so this was to be expected. We can re-run the command, and this time, append the --sudo option, which will tell Ansible that it should run the apt-get command as root, so that the package will be installed correctly.

```
vagrant@mgmt:~$ ansible web1 -m apt -a "name=ntp state=installed" --sudo
web1 | success >> {
    "changed": true,
    "stderr": "",
    "stdout": "Reading package lists...\nBuilding dependency tree...\nReading state information...\nThe following ex
    will be installed:\n  libopts25\nSuggested packages:\n  ntp-doc\nThe following NEW packages will be installed:\n
    ntp\n0 upgraded, 2 newly installed, 0 to remove and 0 not upgraded.\nNeed to get 666 kB of archives.\nAfter this
    1668 kB of additional disk space will be used.\nGet:1 http://archive.ubuntu.com/ubuntu/ trusty/main libopts25 am
    [55.3 kB]\nGet:2 http://archive.ubuntu.com/ubuntu/ trusty/main ntp amd64 1:4.2.6.p5+dfsg-3ubuntu2 [611 kB]\nFetc
    (414 kB/s)\nSelecting previously unselected package libopts25:amd64.\n(Reading database ... 60931 files and dire
    installed.)\nPreparing to unpack .../libopts25_1%3a5.18-2ubuntu2_amd64.deb ...\nUnpacking libopts25:amd64 (1:5.1
    Selecting previously unselected package ntp.\nPreparing to unpack .../ntp_1%3a4.2.6.p5+dfsg-3ubuntu2_amd64.deb .
    (1:4.2.6.p5+dfsg-3ubuntu2) ...\nProcessing triggers for man-db (2.6.7.1-1) ...\nProcessing triggers for ureadahe
    Setting up libopts25:amd64 (1:5.18-2ubuntu2) ...\nSetting up ntp (1:4.2.6.p5+dfsg-3ubuntu2) ...\n * Starting NTP
    ...done.\nProcessing triggers for libc-bin (2.19-0ubuntu6.3) ...\nProcessing triggers for ureadahead (0.100.0-16
}
```

This time we get back a nice green message, saying that the change state is true, with a long message from when the package was installed. If we run the ad-hoc command again, nothing should be done, as the package has already been installed.

```
vagrant@mgmt:~$ ansible web1 -m apt -a "name=ntp state=installed" --sudo
web1 | success >> {
    "changed": false
}
```

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

vagrant@mgmt:~$ cat files/ntp.conf
driftfile /var/lib/ntp/ntp.drift
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable
server 0.ubuntu.pool.ntp.org
server 1.ubuntu.pool.ntp.org
server 2.ubuntu.pool.ntp.org
server 3.ubuntu.pool.ntp.org
server ntp.ubuntu.com
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery
restrict 127.0.0.1
restrict ::1
```

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 gro
web1 | success >> {
    "changed": true,
    "checksum": "f1f51d84bd084c9acbc1a1827b70860db2117ae4",
    "dest": "/etc/ntp.conf",
    "gid": 0,
    "group": "root",
    "md5sum": "5b7b1e1e54f33c6948335335ab03f423",
    "mode": "0644",
    "owner": "root",
    "size": 504,
    "src": "/home/vagrant/.ansible/tmp/ansible-tmp-1423790069.85-18596918716046/source",
    "state": "file",
    "uid": 0
}
```

We are returned a green success message, saying that the file was changed, along with a bunch of meta data about the change. We can run the same command again, and this time you will notice that we did not change the file, but just verified it was the correct one.

```
vagrant@mgmt:~$ ansible web1 -m copy -a "src=/home/vagrant/files/ntp.conf dest=/etc/ntp.conf mode=644 owner=root gro
web1 | success >> {
    "changed": false,
    "checksum": "f1f51d84bd084c9acbc1a1827b70860db2117ae4",
    "dest": "/etc/ntp.conf",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "path": "/etc/ntp.conf",
    "size": 504,
    "state": "file",
    "uid": 0
}
```

Stopping, Starting, and Restarting Services

So far, we have installed a package, and deployed a configuration file, but what about stopping, starting, and restarting services? Well, there is a module for that too. Lets run, ansible web1 -m service -a name=ntp state=restarted. At this point you should know what it happening here. We are using the service module, to restart the ntp process, over on web1. This is a pretty common pattern for deploying application stacks to remote machines, install a bunch of packages, push out your configuration files, then restart the relevant services.

```
vagrant@mgmt:~$ ansible web1 -m service -a "name=ntp state=restarted"
web1 | success >> {
    "changed": true,
    "name": "ntp",
    "state": "started"
}
```

We can also re-run the command, and this time, since we used the restarted state, it will always change, but you can play around with the different states.

```
vagrant@mgmt:~$ ansible web1 -m service -a "name=ntp state=restarted"
web1 | success >> {
    "changed": true,
    "name": "ntp",
    "state": "started"
}
```

Ah-hoc Shell Commands

To be honest, you are hardly even going to use ad-hoc commands to install packages, deploy files, or restart service, as this workflow fits much better into Playbooks. I just wanted to illustrate that you had the option via ad-hoc commands, if you wanted too, and it works well in examples to build up our knowledge of how Ansible functions. There is however, one ad-hoc command type that will likely get lots of mileage out of, and that it running shell commands via Ansible.

I am just going to type the command, then we can review what it does, ansible all -m shell -a uptime. You can probably already guess what is going to happen, but lets work through it. We are using an ad-hoc ansible command, targeting all nodes in our hosts inventory, using the shell module, which allows us to run remote commands, and we want to execute the uptime command.

```
vagrant@mgmt:~$ ansible all -m shell -a "uptime"
web1 | success | rc=0 >>
 01:28:42 up  4:40,  1 user,  load average: 0.00, 0.01, 0.05

web2 | success | rc=0 >>
 01:28:41 up  4:40,  1 user,  load average: 0.01, 0.02, 0.05

lb | success | rc=0 >>
 01:28:41 up  4:41,  1 user,  load average: 0.00, 0.01, 0.05
```

We are returned the command output from all machines in our hosts inventory, you could limit this to various groups, or individual machines too. Lets change the command from uptime, to something like uname -a, maybe we want to see what kernel versions are running.

```
vagrant@mgmt:~$ ansible all -m shell -a "uname -a"
web2 | success | rc=0 >>
Linux web2 3.13.0-35-generic #62-Ubuntu SMP Fri Aug 15 01:58:42 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux

lb | success | rc=0 >>
Linux lb 3.13.0-35-generic #62-Ubuntu SMP Fri Aug 15 01:58:42 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux

web1 | success | rc=0 >>
Linux web1 3.13.0-35-generic #62-Ubuntu SMP Fri Aug 15 01:58:42 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

Or, maybe we want to reboot all the web nodes. You get the idea. Say for example, that there is a cooling failure in your data centre, and that you need to 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

What I have shown you so far, has mostly been an illustration of what you can do at the command line using Ansible. But, to be honest, this can be cumbersome as you are essentially still doing things manually by way of typing things out. This is still an improvement over manually logging into each of these client nodes one by one, but we can do

better, by moving this package, configuration file, and service pattern, into a Playbook. We already had a limited interaction with Playbooks by deploying our ssh authorized key to our client nodes, but lets check out what deploying a package, configuration file, and toggling the service state looks like.

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.

Again, we are going to disable gathering facts about these machines, as we are not going to use them just yet. I will show you what this does in a minute though.

Next, we get into the tasks section, and this is where things should start to look familiar, as we pretty much ran all of this on the command line already. This first task definition installs the ntp package. Task definitions will typically span two lines, the first line will be a name, or title of the task, then you will have the module for task the you want to run on the second line. In our case, we are using the apt module, which we discussed earlier, and pass along an argument string, making sure the package is installed, and that the apt cache is up to date.

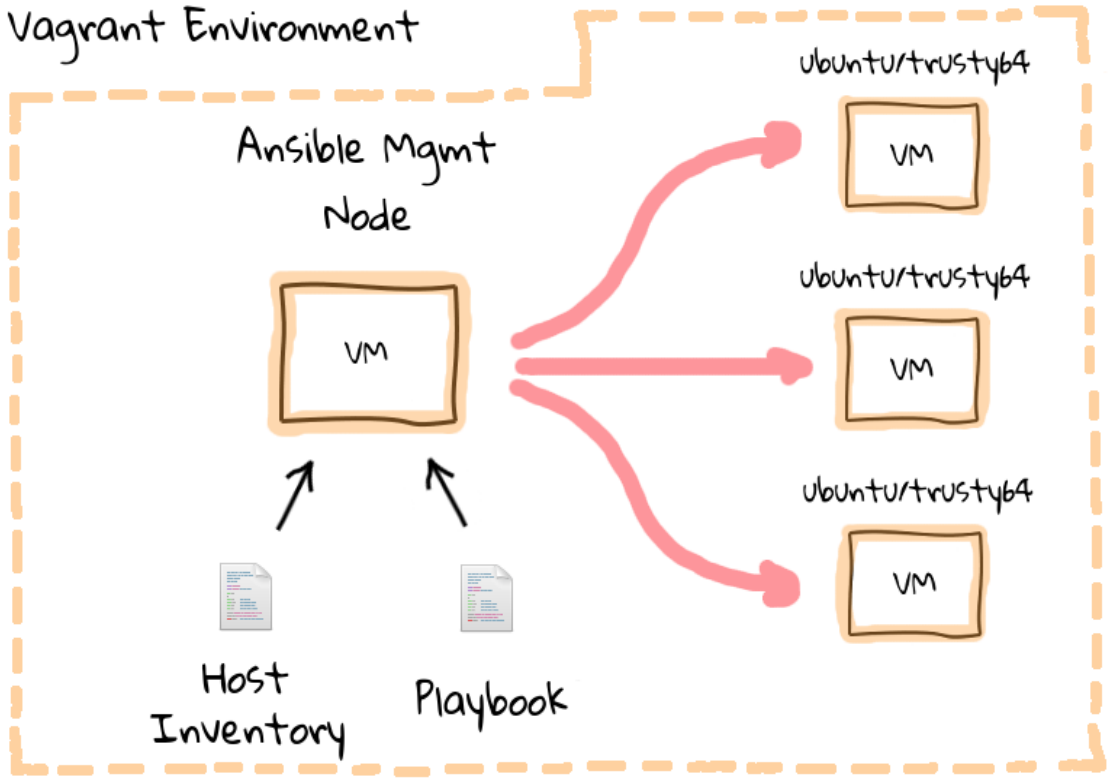
The next block looks pretty much the same, we have our task title, specific our copy module, define the source file, destination file, along with the file modes. Something new here, is this, notify restart ntp bit. This is actually pretty cool. So, you might remember back when we were running our ad-hoc commands, that you can run the same command over and over, and things will only change if something needs to be changed. So, if this file has been detected to change, on a copy over via a ansible run, it will notify this handler. The handlers job, is to run some type of

action, in this case, we are going to restart ntp. You can actually read all about handlers on the Ansible documentation site, but basically these are used as a type of trigger to fire off an event when something happens. The idea here, is that say for example we update the ntp configure file, we can run this playbook, and it will know that ntp needs to be restarted, via this handler, and it will restart the process to get our new configure file settings. Handlers have the same formatting as regular tasks, just that they can be called from other tasks, to do interesting things.

The final task here, is that we want to make sure ntp is started, chances are it will be, but lets just make sure anyway. This is basically just a more advanced version of the package, configuration file, and service pattern that we reviewed, using the ad-hoc commands earlier, but in playbook form. We can run it by typing, ansible-playbook, then the playbook name, so e45-ntp-install.yml.

```
vagrant@mgmt:~$ ansible-playbook e45-ntp-install.yml

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

TASK: [install ntp] ************************************************************
ok: [web1]
changed: [web2]
changed: [lb]

TASK: [write our ntp.conf] *****************************************************
ok: [web1]
changed: [web2]
changed: [lb]

TASK: [start ntp] **************************************************************
ok: [web2]
ok: [web1]
ok: [lb]

NOTIFIED: [restart ntp] ********************************************************
changed: [web2]
changed: [lb]

PLAY RECAP *********************************************************************
lb                         : ok=4    changed=3    unreachable=0    failed=0
web1                       : ok=3    changed=0    unreachable=0    failed=0
web2                       : ok=4    changed=3    unreachable=0    failed=0
```

Okay, so our playbook has run through the three machines in our hosts inventory, we could have targeted a narrow group of machines too, but it is more fun to work with a group of client nodes. The pattern here, is really useful for laying down a common build, or standard package set, across a large number of machines. Say for example, that you are running a data centre with a few hundred machines, you will likely want common dns settings, ntp time settings, syslog going to a central server, keys and accounts, security tweaks, all these sorts of things. Ansible is perfectly suited for deploying these types things.



So, lets work through what the output is telling us here. First, we installed the ntp package across our nodes, and you will notice that we already installed ntp via the ad-hoc commands earlier onto web1, so nothing was done there, but for web2 and lb, ntp was installed. Same thing happened when we deployed the configuration file, web1 has it, but web2 and lb were updated. Next, we make sure ntp is started. Finally, web2 and lb, tripped off the notify handler because the config file changed, so we trigger a restart of ntp. The play recap gives us a tally of the state changes for each of our client nodes. For example, web1 was already up to date via our ad-hoc commands, so nothing was changed, but lb and web2 were changed. This is useful for seeing at a glace what was updated, then you can go back and look through the output, if you notice something unexpected.

As usual, lets re-run the ansible-playbook e45-ntp-install.yml command, to see what the output looks like when nothing changes. As you can see from the recap, nothing was changed, and our environment conforms to what the playbook defines. Think about that for a minute. This is vastly superior to doing things manually, as we can quickly turn machines into what we want, verify they are configured like we want, but not only that, we also have a self documenting playbook, which outlines, step by step, how it was created. This is light years ahead of doing things manually. We have not even got into the really cool stuff yet, but this just kind of outlines the work flow, and benefits that using configuration management can provide.

```
vagrant@mgmt:~$ ansible-playbook e45-ntp-install.yml

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

TASK: [install ntp] ************************************************************
ok: [web2]
ok: [lb]
ok: [web1]

TASK: [write our ntp.conf] *****************************************************
ok: [lb]
ok: [web2]
ok: [web1]

TASK: [start ntp] **************************************************************
ok: [web2]
ok: [web1]
ok: [lb]

PLAY RECAP *********************************************************************
lb                         : ok=3    changed=0    unreachable=0    failed=0
web1                       : ok=3    changed=0    unreachable=0    failed=0
web2                       : ok=3    changed=0    unreachable=0    failed=0
```

Configuration Change Example

I thought it might be useful, to create an example based around handlers, as this is something that is quite common in the workflow of a Sysadmin. You will have a change request come in, where you need to tweak a configuration file, then update a bunch of nodes with the latest configuration file. So, lets work through what something like that would look like using Ansible.

We already deployed the ntp.conf file, from the files sub directory on our management node, but say for example, we wanted to change the ntp server that each of our client nodes uses, then deploy that change out across our infrastructure. Lets open the ntp.conf file using vi. So, lets say we wanted to replace this server block here, with someone a little more specific to our environment.

```
vagrant@mgmt:~$ vi files/ntp.conf
driftfile /var/lib/ntp/ntp.drift
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable
server 0.ubuntu.pool.ntp.org
server 1.ubuntu.pool.ntp.org
server 2.ubuntu.pool.ntp.org
server 3.ubuntu.pool.ntp.org
server ntp.ubuntu.com
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery
restrict 127.0.0.1
restrict ::1
```

For example, I live in Canada, so I went over to ntp.org, and found the server list for my country, and as it says here, just copy and paste the following into your ntp.conf file. Lets just copy these, then jump back to our ntp.conf file, and paste them in here. Okay, so we have our updated ntp.conf file, now what?

```
vagrant@mgmt:~$ vi files/ntp.conf
driftfile /var/lib/ntp/ntp.drift
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable
server 0.ca.pool.ntp.org
server 1.ca.pool.ntp.org
server 2.ca.pool.ntp.org
server 3.ca.pool.ntp.org
server ntp.ubuntu.com
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery
restrict 127.0.0.1
restrict ::1
```

Well, lets just use our e45-ntp-install.yml playbook again, as it has all the logic we are looking for. Mainly that it deploys our configuration file, and if it has changed, then it triggers the handler, which will restart the ntp service.

Lets give it a shot, by running ansible-playbook 45-ntp-install.yml. Cool. So, you can see that our updated ntp.conf file was deployed, and since it changed, our handler was notified that it needed to restart ntp, so that it would reload the configuration file. If you are not using configuration management, something as simple as this can be a real pain point, but Ansible can turn this weakness into a strength.

```
vagrant@mgmt:~$ ansible-playbook e45-ntp-install.yml

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

TASK: [install ntp] ************************************************************
ok: [lb]
ok: [web1]
ok: [web2]

TASK: [write our ntp.conf] *****************************************************
changed: [web2]
changed: [web1]
changed: [lb]

TASK: [start ntp] **************************************************************
ok: [web2]
ok: [web1]
ok: [lb]

NOTIFIED: [restart ntp] ********************************************************
changed: [lb]
changed: [web1]
changed: [web2]

PLAY RECAP *********************************************************************
lb                         : ok=4    changed=2    unreachable=0    failed=0
web1                       : ok=4    changed=2    unreachable=0    failed=0
web2                       : ok=4    changed=2    unreachable=0    failed=0
```

Gathering Facts

We are going to finish off this episode talking about facts, variables, and templates, as they are all interrelated. It really helps to have a base knowledge of what Ansible is, and how playbooks function before exploring the topic, so that is why I have waited to talk about them.

What are facts? Well, we briefly talked about them when looking at our examples playbooks, because we turned fact gathering off, but when an Ansible playbook run executes against a remote machine, a task will launch, and gather facts about that remote machine. These can be used for all types of things, but let me show you what they look like, before we go any further.

We can run an ad-hoc version, like this, ansible web1 -m setup, then we are just going to pipe the output into less, so that we can page through it. So, we are running the setup module against web1.

```
vagrant@mgmt:~$ ansible web1 -m setup | less
web1 | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15",
            "10.0.15.21"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::a00:27ff:fe2c:1424",
            "fe80::a00:27ff:fea4:bab"
        ],
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "12/01/2006",
        "ansible_bios_version": "VirtualBox",
        "ansible_cmdline": {
            "BOOT_IMAGE": "/boot/vmlinuz-3.13.0-35-generic",
            "console": "ttyS0",
            "ro": true,
            "root": "UUID=9c0eb3f7-22d4-4252-85bd-ff3096d10068"
        },
```

Moments later, we are presented with tons of facts about this remote machine, things like the IP addresses assigned, boot environment information, disk partitions, detailed network configuration information, things like MAC addresses, and the list goes on and on. So, what would you use these type of facts for? Well, in parts three and four of this episode series, we are going to be building a dynamic load balancer configuration file, which will use IP address facts gathered from all our web server nodes, so that we can direct traffic to them.

Lets jump back to the command line, and try some more ad-hoc commands, by using the filter argument passed to the setup module. Lets say for example, that you are running a playbook against a cluster of machines, made up of Ubuntu and CentOS boxes. You might want to do something that says, if the distribution is Ubuntu, run the apt module, but if it is CentOS, then use the yum module. Simple things like that.

Lets add the argument option, and pass in, filter=ansible_distribution. So, you might want to run this across a bunch of machines to gather some data about them too, say for example that you are trying to get an inventory, based off some fact.

```
vagrant@mgmt:~$ ansible web1 -m setup -a "filter=ansible_distribution"
web1 | success >> {
    "ansible_facts": {
        "ansible_distribution": "Ubuntu"
    },
    "changed": false
}
```

You can also grab facts using the star character. Rerunning the command, you now see all facts matching that name. Before we did and exact match, but the star allows us to grab everything with that name, which is nice if you are not exactly sure what you are looking for.

```
vagrant@mgmt:~$ ansible web1 -m setup -a "filter=ansible_distribution*"
web1 | success >> {
    "ansible_facts": {
        "ansible_distribution": "Ubuntu",
        "ansible_distribution_major_version": "14",
        "ansible_distribution_release": "trusty",
        "ansible_distribution_version": "14.04"
    },
    "changed": false
}
```

Or, like we are going to do in the next episode, use the template module, to gather IP addresses from our web servers into a load balancer configuration file. Lets run the filter against, ansible all ipv4 addresses, and you can see a list of the addresses web1 is listening on. I do not want to cover this too much here, but I just wanted to introduce the concept, and we can develop it further in parts three and four.

```
vagrant@mgmt:~$ ansible web1 -m setup -a "filter=ansible_all_ipv4_addresses"
web1 | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15",
            "10.0.15.21"
        ]
    },
    "changed": false
}
```

Templates and Variables

Okay, the final thing that I wanted to show you is templates. I have created a playbook, called e45-ntp-template.yml, which is actually very similar to our e45-ntp-install.yml playbook. Lets just cat the contents and work through what is happening.

```
vagrant@mgmt:~$ cat e45-ntp-template.yml
---
- hosts: all
  sudo: yes
  gather_facts: no
  vars:
    noc_ntpserver: 0.ca.pool.ntp.org

  tasks:

  - name: install ntp
    apt: name=ntp state=installed update_cache=yes

  - name: write our ntp.conf
    template: src=/home/vagrant/files/ntp.conf.j2 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
```

So, we are targeting all node, using sudo, and not gathering any facts. Gathering facts does not actually hurt anything, it is actually turned on by default. I only turned it off because it adds an extra couple seconds to the playbook run. These next two lines are something new, variables are basically custom facts that you can define. So, in this case, I am defining a fact called, noc_ntpserver. This might be useful if you had an internal ntp server that you wanted to point your machines at. The only difference between this playbook and the ntp-install playbook, is that we are using the template module, instead of the copy module. The module arguments even look the same, with the excepting of this j2 extension here. J2 stands for Jinja2, and it is a python template engine, Ansible is also written in python in case you did not know.

The template is also included in the episode supporting materials, under the files directory, named ntp.conf.j2. Lets cat that file and see what it looks like. So, the first line up here, looks like a comment, I just added this as the ansible managed fact will output a bunch of data about how this file was generated, we will see this in action after the playbook run. You can include facts or variables, using the fact name, and then the two curly braces around it.

```
vagrant@mgmt:~$ ls -l files/
total 8
-rw-r--r-- 1 vagrant vagrant 466 Feb 13 01:53 ntp.conf
-rw-r--r-- 1 vagrant vagrant 417 Jan 24 18:29 ntp.conf.j2

vagrant@mgmt:~$ cat files/ntp.conf.j2
# {{ ansible_managed }}
driftfile /var/lib/ntp/ntp.drift
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable
server {{ noc_ntpserver }}
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery
restrict 127.0.0.1
restrict ::1
```

Down here, you will notice our server section has been replaced, by the noc_ntpserver custom variable. This will take the variable from our e45-ntp-template.yml playbook, and insert our predefined value. This is useful for moving configuration out of the files, and into the playbook, and it is a bit of a personal preference. The idea is to add custom variables into the playbook, rather than digging through the configuration files. It is not uncommon to see tens of custom variables in more advanced playbooks.

Lets just have a peak at our playbook again, you can see our custom fact defined, and at the top of the screen, you can see where it will be replaced via the server noc_ntpserver jinja2 template entry.

Lets go ahead and see what this does, by running the e45-ntp-template.yml playbook. You should recognize what each of these steps is doing by now, but our ntp.conf file was updated on the remote nodes, and a notification trigger fired to restart the ntp service.

```
vagrant@mgmt:~$ ansible-playbook e45-ntp-template.yml

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

TASK: [install ntp] ************************************************************
ok: [web1]
ok: [lb]
ok: [web2]

TASK: [write our ntp.conf] *****************************************************
changed: [web1]
changed: [lb]
changed: [web2]

TASK: [start ntp] **************************************************************
ok: [lb]
ok: [web2]
ok: [web1]

NOTIFIED: [restart ntp] ********************************************************
changed: [web2]
changed: [web1]
changed: [lb]

PLAY RECAP *********************************************************************
lb                         : ok=4    changed=2    unreachable=0    failed=0
web1                       : ok=4    changed=2    unreachable=0    failed=0
web2                       : ok=4    changed=2    unreachable=0    failed=0
```

This will likely make the most sense, if we just log into web1, and have a peak at the /etc/ntp.conf file. What is so cool about this, is that we have been installing packages, tweaking configure files, and restarting services, without ever logging into these machines. Pretty cool.

So, lets just cat the /etc/ntp.conf file. Right away, you can see the comment at the top of the file, that used to be our double curly braced ansible_managed fact. It gives useful information, like that the file was generated by Ansible, where the template lives, when it was modified, who modified it, and from what machine. Then down here, you can see our custom server entry, with the noc_ntpserver fact, which we defined in the playbook.

```
vagrant@mgmt:~$ ssh web1

vagrant@web1:~$ cat /etc/ntp.conf
# Ansible managed: /home/vagrant/files/ntp.conf.j2 modified on 2015-01-24 18:29:35 by vagrant on mgmt
driftfile /var/lib/ntp/ntp.drift
statistics loopstats peerstats clockstats
filegen loopstats file loopstats type day enable
filegen peerstats file peerstats type day enable
filegen clockstats file clockstats type day enable
server 0.ca.pool.ntp.org
restrict -4 default kod notrap nomodify nopeer noquery
restrict -6 default kod notrap nomodify nopeer noquery
restrict 127.0.0.1
restrict ::1
```

So, that is basically a crash course on facts, variables, and templates. Lets exit web1 and head back to the management node.

Cleaning Up

I though we could come full circle, and clean up our environment, by removing ntp from the client nodes. The final example for this episode, is the e45-ntp-remove.yml playbook, lets just quickly take a look at what it does.

```
vagrant@mgmt:~$ cat e45-ntp-remove.yml
---
- hosts: all
  sudo: yes
  gather_facts: no

  tasks:

  - name: remove ntp
    apt: name=ntp state=absent
```

At this point, hopefully you understand everything that is happening in here. Our lone task, is making sure the ntp package is absent on all client nodes. Lets execute it, by running ansible-playbook e45-ntp-remove.yml.

```
vagrant@mgmt:~$ ansible-playbook e45-ntp-remove.yml

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

TASK: [remove ntp] ************************************************************
changed: [web2]
changed: [lb]
changed: [web1]

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
```

So, we just removed ntp from all client machines, and a rerun of the same playbook shows that nothing was done, meaning that our ntp package was been wiped off our client nodes.

```
vagrant@mgmt:~$ ansible-playbook e45-ntp-remove.yml

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

TASK: [remove ntp] ************************************************************
ok: [lb]
ok: [web2]
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
```

That's a Wrap

By now, hopefully you have a pretty good understanding of what Ansible is, how it works, and what you might use it for. I mentioned back in part one, of this episode series, that the bar is pretty low for working with Ansible. I think we also worked through some examples, where configuration management really excels when compared to doing things manually. Ansible is not a silver bullet, in that you still need to create playbooks, and work through problems, but it does make things much easier as time goes by, and you will develop an arsenal of useful playbooks, and one off commands to solve your problems.

Well, that wraps up this monster of an episode, and we covered tons of material, so hopefully you found it a worthwhile watch. So, if you are interested in playing around with Ansible, you have the supporting Vagrantfile, bootstrap script, and a bunch of examples to work through. Be sure to check out part three of this series, where we deploy a web cluster, and configure a haproxy load balancer. Then in part four, we will do several rolling website deployments across our cluster.