Episode #46 - Configuration Management with Ansible (Part 3/4)

About Episode - Duration: 28 minutes, Published: Mar 26, 2015

In this episode, we are going create a load balanced web cluster by way of our Vagrant environment. The goal is to demonstrate how Ansible can solve real world problems by building up infrastructure from scratch.

Download: mp4 (https://s3.amazonaws.com/sysadmincasts.com/static/videos/46-configuration-management-with-ansible-part-3-4.mp4) or webm (https://s3.amazonaws.com/sysadmincasts.com/static/videos/46-configuration-management-with-ansible-part-3-4.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 #45 - Learning Ansible with Vagrant (Part 2/4) (/episodes/45-learning-ansible-with-vagrant-part-2-4)
- Episode #47 - Zero-downtime Deployments with Ansible (Part 4/4) (/episodes/47-zero-downtime-deployments-with-ansible-part-4-4)
- Ansible Documentation (http://docs.ansible.com/index.html)
- nginx (http://nginx.org/en/)
- HAProxy (http://www.haproxy.org/)
- Ansible - Module Index (http://docs.ansible.com/modules_by_category.html)
- Ansible - Playbooks (http://docs.ansible.com/playbooks.html)
- Ansible - Ping Module (http://docs.ansible.com/ping_module.html)
- Ansible - Template Module (http://docs.ansible.com/template_module.html)
- Ansible - Lineinfile Module (http://docs.ansible.com/lineinfile_module.html)

- Ansible - Loops (http://docs.ansible.com/playbooks_loops.html)
- Ansible - Setup Module (http://docs.ansible.com/setup_module.html)
- Ansible - Roles (http://docs.ansible.com/playbooks_roles.html)

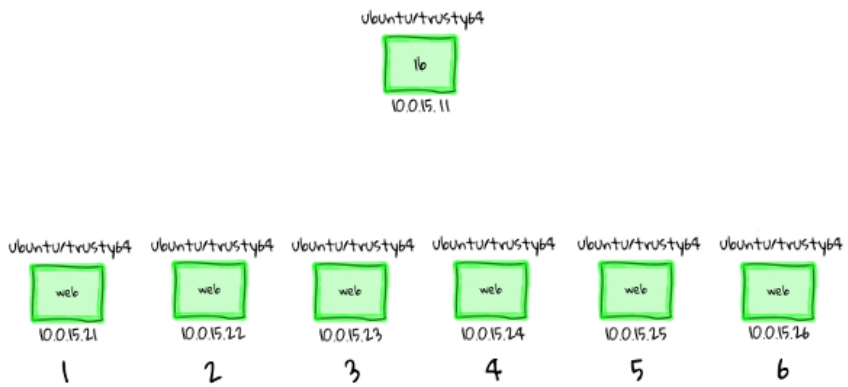
In this episode, we are going create a load balanced web cluster using Vagrant and Ansible. The goal is to show how Ansible can solve real problems by building up complex infrastructure from scratch.

Series Recap

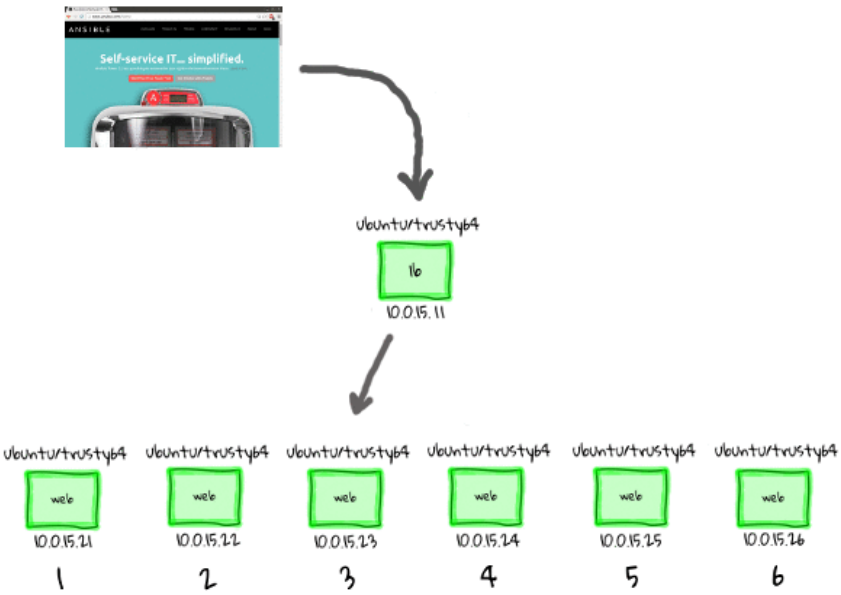
Before we dive in, lets quickly review what this episode series is about. In part one, episode #43, we looked at what Ansible is at a high level, basically a comparison of doing things manually, versus using configuration management. Part two, episode #45, served as a bit of an Ansible crash course, 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.

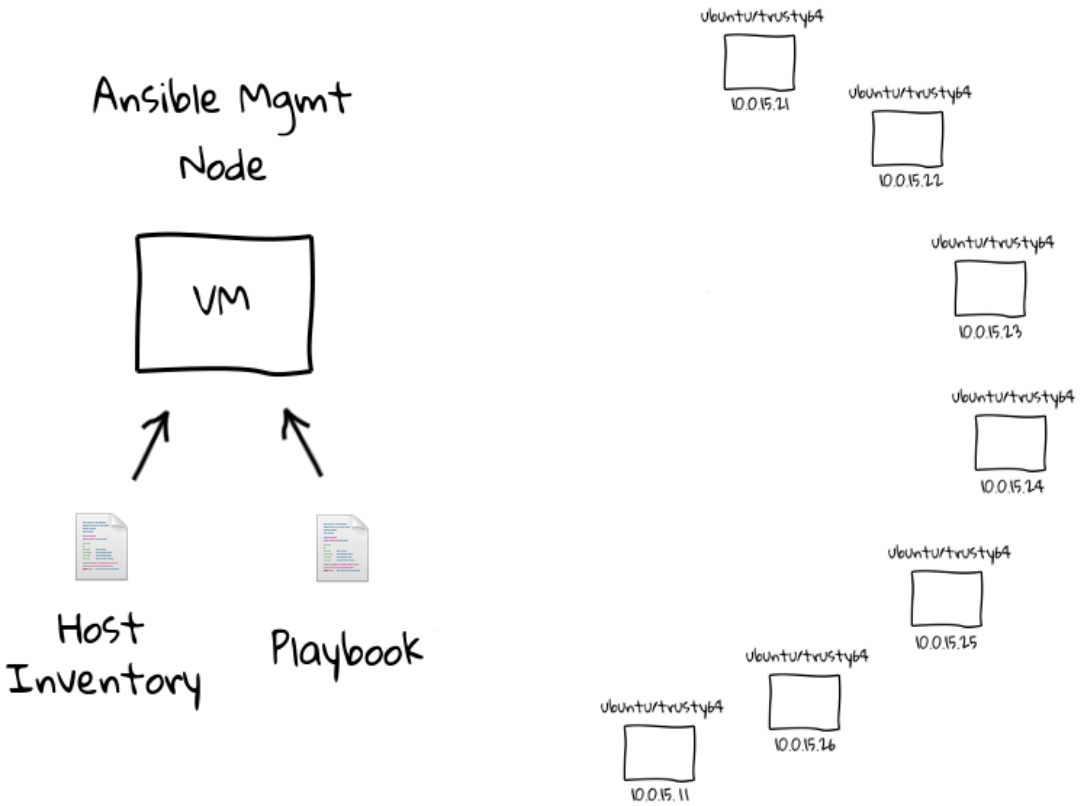


We will then fire up a web browser and start sending requests to the load balancer. We will watch, as our requests cycle through the six web servers, while we hit refresh in the browser. This is not only a pretty cool demo, but we will also learn some new things about Ansible, as we work to get this configured.



How Do We Build It

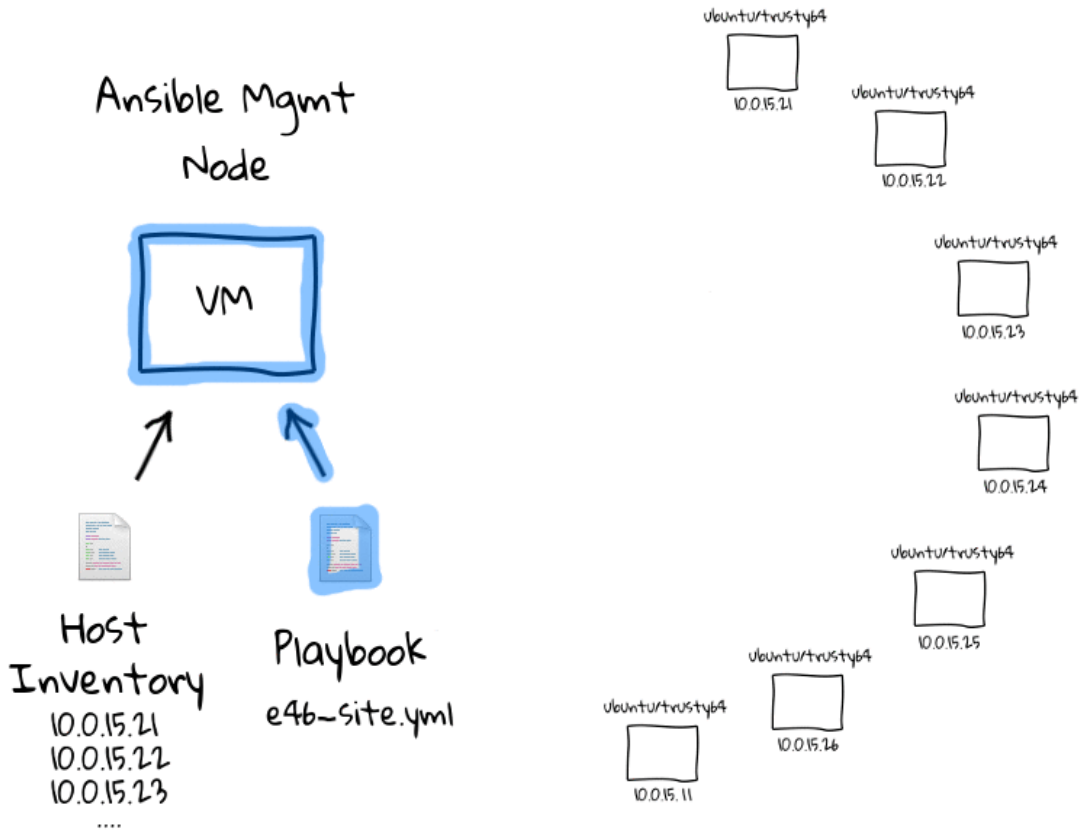
So, how are we going to build this? Well, lets work through the process at a high level, before we jump to the command line and make it happen. We are going to pick up from where we left our Vagrant environment in part two of this series. We had installed, and configured, an Ansible management node, with hosts inventory, and supporting episode playbooks. We also had two web nodes, but this time around, we will bump that number up to six through the Vagrantfile. I should stress, that this is not required, you can leave it at two. I am just doing this for dramatic effect, because it is fun to see Ansible work across a bunch of machines. Finally, lets not forget about the load balancer. So, our updated Vagrant environment will have a total of eight virtual machines, an Ansible management node, a load balancer, and six web servers.



With the Vagrantfile updated, and our additional web nodes launch, we can then log into the management node, where we can update the hosts inventory, deploy our ssh key for password less access, and test connectivity using the ping module within our Vagrant environment.

The tasks so far, are basically to on board the new nodes into our environment, and make sure Ansible can talk to them. Once we have all that sorted out, we will review the e46-site playbook. This e46-site playbook contains all the interesting bits, which allows to configure the load balanced web cluster, in just a little over 60 lines of text. You will likely recognize the package, file, and service pattern, which will allow us to configure the load balancer and six web nodes. I think you will be surprised, at just how simple it is, to configure an environment like this.

Okay, so armed with this script, we will execute ansible-playbook e46-site on the management node. Ansible will read in our e46-site playbook, check out the hosts inventory to see which nodes it needs to run commands against, then at this point, it is ready to go. Ansible will connect out to all of these machines in parallel, through an ssh trust, and turn our default virtual machines into the configure that we have defined via the playbook.



What I find so neat about this, is that as far as Ansible is concerned, it is just doing what you told it, but the end result from this one command is and environment that looks like this. To round out this episode we will review the haproxy load balancer interface too, so that we can get some statistics about which servers are getting traffic, and it is pretty cool to get a behind the scenes look at this.

Lets Build It

Okay, I have probably talked long enough about what we are going to build, and how we are going to build it. So, lets hop over to the command line, and actually do it.

First, we are going to change into the e45 directory. This is the project directory, where we extracted the supporting examples archive used back in part two. If you do not know what this is, make sure you watch part two of this episode series.

```
[~]$ cd e45

[e45]$ ls -l
-rw-rw-r-- 1 jw jw  676 Feb 12 00:12 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 Mar  4 14:56 Vagrantfile
```

Modifying The Vagrant Environment

We can check the status of our Vagrant environment by running, vagrant status, and as you can see, we still have our four virtual machines running from part two. You will notice that we only have two web servers here, so lets open up the Vagrantfile in an editor, and bump that number up to six.

```
[e45]$ vagrant status
Current machine states:

mgmt                      running (virtualbox)
lb                        running (virtualbox)
web1                      running (virtualbox)
web2                      running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

Just a little recap, but we have three blocks of code in here, one for our management node, one for the load balancer, and the final one here is for our web nodes. We simply need to update the number two here, to the number six, and that will add our additional web nodes. Again, this is not required, but I think it makes for a better demo. There is one hidden bonus too, which is that we can work through the motions of adding brand new machines into our Ansible environment. Okay, so lets save and exit out of here.

```
# Vagrantfile web server code block
#
# https://docs.vagrantup.com/v2/vagrantfile/tips.html
(1..6).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
```

With the Vagrantfile updated, lets run vagrant status again, and we should see our new web nodes. Great, so we have eight nodes total now in our Vagrant environment, the four original ones, and our four new web servers, web three, four, five, and six. You will also notice that they are in the not created state.

```
[e45]$ vagrant status
Current machine states:

mgmt                      running (virtualbox)
lb                        running (virtualbox)
web1                      running (virtualbox)
web2                      running (virtualbox)
web3                      not created (virtualbox)
web4                      not created (virtualbox)
web5                      not created (virtualbox)
web6                      not created (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

So, lets run vagrant up, and this will launch those new machines into our vagrant environment so that we have access to them. This took about three and a half minutes in real-time, but I have just sped up the video, just to make things a little quicker.

```
[e45]$ vagrant up
...
Bringing machine 'web3' up with 'virtualbox' provider...
Bringing machine 'web4' up with 'virtualbox' provider...
Bringing machine 'web5' up with 'virtualbox' provider...
Bringing machine 'web6' up with 'virtualbox' provider...
....
```

Now, we can run, vagrant status again, just to verify everything is in an up and running state. Great, looks good.

```
[e45]$ vagrant status
Current machine states:

mgmt                      running (virtualbox)
lb                        running (virtualbox)
web1                      running (virtualbox)
web2                      running (virtualbox)
web3                      running (virtualbox)
web4                      running (virtualbox)
web5                      running (virtualbox)
web6                      running (virtualbox)

This environment represents multiple VMs. The VMs are all listed
above with their current state. For more information about a specific
VM, run `vagrant status NAME`.
```

Management Node Tweaks

Lets connect to the management node by running, vagrant ssh mgmt.

```
[e45]$ vagrant ssh mgmt
Welcome to Ubuntu 14.04.1 LTS (GNU/Linux 3.13.0-35-generic x86_64)

 * Documentation:  https://help.ubuntu.com/

 System information disabled due to load higher than 1.0

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.


vagrant@mgmt:~$
```

Nothing has changed on the management node since part two, in that we still have all our examples here, along with the Ansible config overrides and inventory files.

```
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

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 will notice that our inventory file only has web one and two right now, these others are commented out, so lets uncomment web three, four, five, and six. We can do that by opening the inventory file in vi. Lets just scroll down here and uncomment these lines. Now Ansible should recognize these four new nodes as part of the web group. I am just going to save and head back to the command line.

```
vagrant@mgmt:~$ vi inventory.ini
```

I have probably mentioned this before, but when I make a change, I generally like to verify that it worked. We can do that by checking to see, if the file looks like, what we think it should look. This has saved me countless times, while deploying changes, or working on helpdesk tickets, and I think it is just a good habit to develop.

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

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

Verifying Network Connectivity

We should actually verify our management node has connectivity to these new web servers too. Lets test this out by pinging web3, then web4, web5, and finally web6. Okay, so the management node can at least talk to these machines on a network level, and name resolution is working via our /etc/hosts entries make in part two.

```
vagrant@mgmt:~$ ping web3
PING web3 (10.0.15.23) 56(84) bytes of data.
64 bytes from web3 (10.0.15.23): icmp_seq=1 ttl=64 time=1.35 ms
64 bytes from web3 (10.0.15.23): icmp_seq=2 ttl=64 time=0.704 ms
--- web3 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.704/1.031/1.359/0.329 ms

vagrant@mgmt:~$ ping web4
PING web4 (10.0.15.24) 56(84) bytes of data.
64 bytes from web4 (10.0.15.24): icmp_seq=1 ttl=64 time=0.826 ms
64 bytes from web4 (10.0.15.24): icmp_seq=2 ttl=64 time=0.750 ms
--- web4 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1000ms
rtt min/avg/max/mdev = 0.750/0.788/0.826/0.038 ms

vagrant@mgmt:~$ ping web5
PING web5 (10.0.15.25) 56(84) bytes of data.
64 bytes from web5 (10.0.15.25): icmp_seq=1 ttl=64 time=1.02 ms
64 bytes from web5 (10.0.15.25): icmp_seq=2 ttl=64 time=0.585 ms
--- web5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.585/0.802/1.020/0.219 ms

vagrant@mgmt:~$ ping web6
PING web6 (10.0.15.26) 56(84) bytes of data.
64 bytes from web6 (10.0.15.26): icmp_seq=1 ttl=64 time=0.941 ms
64 bytes from web6 (10.0.15.26): icmp_seq=2 ttl=64 time=0.507 ms
--- web6 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.507/0.724/0.941/0.217 ms
```

Configuring SSH Access

Lets configure password less access into these new web servers. You might remember that back in part two, we used ssh-keyscan to acknowledge ssh fingerprint authenticity in bulk, and pipe these keys into the known_hosts file.

```
vagrant@mgmt:~$ ls -l .ssh/
total 16
-rw------- 1 vagrant vagrant  466 Feb 23 22:15 authorized_keys
-rw------- 1 vagrant vagrant 1675 Feb 23 22:20 id_rsa
-rw-r--r-- 1 vagrant vagrant  394 Feb 23 22:20 id_rsa.pub
-rw-rw-r-- 1 vagrant vagrant 2318 Feb 23 22:21 known_hosts
```

Then, we deployed the vagrant users public RSA key out to these remote hosts, via the e45-ssh-addkey playbook. Well, since we just launched some new nodes, we will need to repeat this process. Luckily this goes pretty quickly. Lets run, ssh-keyscan web3 web4 web5 web6, then redirect the output into .ssh/known_hosts.

```
vagrant@mgmt:~$ ssh-keyscan web3 web4 web5 web6 >> .ssh/known_hosts
# web4 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# web3 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# web3 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# web4 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# web5 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# web5 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# web6 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
# web6 SSH-2.0-OpenSSH_6.6.1p1 Ubuntu-2ubuntu2
```

With that done, lets push out the vagrant users public RSA key, to web3 web4 web5 web6, through the use of the e45-ssh-addkey playbook. Lets run, ansible-playbook e45-ssh-addkey, and since this is the first time we are connection to our new web nodes, we need to tack on the ask pass option, since we do not have password less access configured yet. The password is just vagrant.

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

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

TASK: [install ssh key] *******************************************************
changed: [web5]
ok: [web1]
changed: [web4]
ok: [web2]
changed: [web3]
changed: [web6]
ok: [lb]

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
web3                       : ok=1    changed=1    unreachable=0    failed=0
web4                       : ok=1    changed=1    unreachable=0    failed=0
web5                       : ok=1    changed=1    unreachable=0    failed=0
web6                       : ok=1    changed=1    unreachable=0    failed=0
```

The output should look familiar, as we have run these types of commands before, but lets work through what is happening here. One task was run against all nodes, and that was to install a ssh key. You can see that web1, web2, and the load balancer already had it, so nothing was done. However, the four new web nodes had the key deployed. Then down here, we have the standard playbook recap.

Verify Ansible Connectivity

Now that we have the hosts inventory updated, and pushed out our ssh key, lets verify Ansible can talk to all nodes. Lets run, ansible all -m ping. So, we are running ansible the ad-hoc version, targeting all nodes, specifying that we want to use a module, and using the ping module. Lets me just scroll up here, and as you can see, we get back ping pong messages from all the hosts in our environment. Cool, so this proves that we have connectivity, and that things are in good shape.

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

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

web5 | success >> {
    "changed": false,
    "ping": "pong"
}

web4 | success >> {
    "changed": false,
    "ping": "pong"
}

web3 | success >> {
    "changed": false,
    "ping": "pong"
}

web6 | success >> {
    "changed": false,
    "ping": "pong"
}

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

I know this was a little long winded, but these are problems that you will needs to solve, when you add new capacity into Ansible. There are easier ways of dealing with this though, for example, rather than having a hosts inventory as a flat file, you could use a database. This means that as you add new machines, there could be some automated method that would update the database for you, and Ansible would just know that new machine exists. You could also automate the deployment of the ssh keys too, by adding it when you install the machine, we are using Vagrant images here, but you might use canned OS images. Or, you could have some type of post-install script that would inject the key into new machines as they are launched. You get the idea. So, even though this was a little long winded, there are easier ways, you just need to figure them out.

Environment Recap

So, at this point, we are here. We have the management configured, with the hosts inventory tweaks, and the updated known_hosts entries. We have verified Ansible ping connectivity out to the six web servers, and load balancer too, so that means everything is working at a basic level. The next step is to, review the e46-site playbook, and then run it against our client nodes. So, lets jump back to the command line and do that.

The E46-Site Playbook

Just going to review the e46-site playbook using less, as it is a little longer than will fit on the screen, then we can scroll through the various sections.

```
vagrant@mgmt:~$ less e46-site.yml
```

I have broken this playbook into three plays, plays are a new concept that I have not shown you yet, but let me explain how they work. Plays basically allow you to target specific groups of nodes, from within the same playbook, lets just work through this at a high level, and it should make sense pretty quickly.

This first play, targets all nodes in our hosts inventory, this is useful for laying down a common configuration across a fleet of machines. Think about all those standard, or baseline changes that you want all machines to have, this is where those bits would go. Then down here, we have something called the web play, this targets all of our web nodes, for doing things like installing the web server, tweaking configuration files, and starting services. Finally, we have something called the load balancer play, which targets our load balancer.

Lets head back to the top of the file so that we can work through what is happening. I guess the best way to explain this, is that I have only shown you playbooks which have one play, but in this playbook, we have added a couple more.

When you have playbooks with multiple plays, like this first one here, you need to really watch the formatting, and how you indent things, as you will get playbook syntax errors if the spaces are off. In practice, this is not such a huge issue, but just wanted to bring it to your attention.

Okay, so lets work through this play, we are targeting all nodes, using sudo, and turning off fact gathering. Next, we have our tasks section, with a single task defined. The idea here, is that we are going to install the git package across all of our machines. We are not actually going to use this yet, but in part four of this series, we are going to be pulling content from a Github repository, and deploying it as our website content.

```
# common
- hosts: all
  sudo: yes
  gather_facts: no

  tasks:

  - name: install git
    apt: name=git state=installed update_cache=yes
```

Down here, we have our web play. We are targeting all web nodes, using sudo, and you will notice that we have not disabled fact gathering, this means that facts will be gathered for these machines by default. We will use these facts in several templates that I will show you in a minute.

```
# web
- hosts: web
  sudo: yes

  tasks:

  - name: install nginx
    apt: name=nginx state=installed

  - name: write our nginx.conf
    template: src=templates/nginx.conf.j2 dest=/etc/nginx/nginx.conf
    notify: restart nginx

  - name: write our /etc/nginx/sites-available/default
    template: src=templates/default-site.j2 dest=/etc/nginx/sites-available/default
    notify: restart nginx

  - name: deploy website content
    template: src=templates/index.html.j2 dest=/usr/share/nginx/html/index.html

  handlers:

  - name: restart nginx
    service: name=nginx state=restarted
```

Lets move onto the tasks section. So, we are installing nginx, deploying a configuration file, via this template, and then restarting nginx if it changed. This notify restart nginx section, comes in handy if you need to deploy new configuration files down the road, and you just want the service to be restarted, to pick up the new settings. We will also check out these configuration files, right after we walk through the playbook, just so that you get a well rounded idea of what is happening.

Next, we deploy the default nginx site. Again, using a template, and notifying nginx that it needs to be restarted anytime we change this file.

Finally, we deploy the default index.html file, using a template. This is really simple, but in a production environment, you would likely be pulling your website content from a version control system. We will actually look at this type of thing in part four of this episode series, doing rolling updates, and switching between release versions pulled from Github. We have our restart nginx handler down here, which will be trigger, if any of these configuration files change.

The last play, in the e46-site playbook, configures our load balancer. We are targeting all machines in the lb group, there is only one at this point. Saying yes, that we want to execute commands via sudo.

```
# lb
- hosts: lb
  sudo: yes

  tasks:

  - name: install haproxy and socat
    apt: pkg={{ item }} state=latest
    with_items:
    - haproxy
    - socat

  - name: enable haproxy
    lineinfile: dest=/etc/default/haproxy regexp="^ENABLED" line="ENABLED=1"
    notify: restart haproxy

  - name: deploy haproxy config
    template: src=templates/haproxy.cfg.j2 dest=/etc/haproxy/haproxy.cfg
    notify: restart haproxy

  handlers:

  - name: restart haproxy
    service: name=haproxy state=restarted
```

Next, we have the tasks section. This block here is something new, but it basically allows you to iterate over a list of packages, and have them installed, without many duplicate code blocks. Say for example, that you wanted to install 15 packages, you would just add them to the with_items list, and then use this apt pkg item line, kind of like a for loop. I

have linked to the Ansible documentation that covers loops, just check out the links section below, and it has all types of examples. I guess the gist of this, is that we are installing haproxy and the socket cat package, but we could easily install more if needed, just by adding to this list.

Next, we are replacing a line in the /etc/haproxy/defaults file, via the lineinfile module, which will allow us to enable haproxy. We also notify the handler if this changes. Finally, we deploy the haproxy configuration file.

You will likely understand the majority of this, as it is just the package, configuration file, and service pattern that we heavily cover in part two. No matter the configuration management tool you wind up using, you will see this theme over and over, with varying degrees of fanciness.

Okay, so that is pretty much it. We are not doing anything to crazy in here, just installing some package, deploying some configuration files, and making sure things are started.

The E46-Site Templates

Lets have a peak through the templates folder, and get a sense for what these configuration files look like. So, we have the four configuration files here.

```
vagrant@mgmt:~$ ls -l templates/
total 16
-rw-r--r-- 1 vagrant vagrant  318 Feb  9 18:57 default-site.j2
-rw-rw-r-- 1 vagrant vagrant 1243 Feb  9 17:58 haproxy.cfg.j2
-rw-rw-r-- 1 vagrant vagrant  671 Feb 24 02:32 index.html.j2
-rw-rw-r-- 1 vagrant vagrant  912 Feb  9 18:57 nginx.conf.j2
```

Lets start out, by reviewing the nginx server configuration file.

```
vagrant@mgmt:~$ less templates/nginx.conf.j2
```

You will notice that with most of these files, they are actually pretty stock configuration files, but I have made a few modifications. The first one, is to add this ansible managed fact up at the top, you should remember this from part two. The idea is to let anyone who views the file on the end system, to know that it was generated by Ansible, along with metadata about how it was generated. This is a good warning, since if someone changes it, there us a high probability of it being overwritten on the next Ansible run.

```
# {{ ansible_managed }}
```

I am not going to talk to much about the stock nginx configuration, as you can easily read about that, but I have made a couple tweaks down here. I found this blog posting, about how to add a header to each server response, just to indicate which backend server served the request. This might not be so useful in production, but it is pretty nice to debug your environment while testing, and I will show you what this looks like in just a minute.

```
    add_header X-Backend-Server $hostname;
```

Again, this is not a good idea for production, but I have disabled all caching. This will allow us to debug the environment, without any client or server side caching issues. So, that is pretty much it, nothing too crazy, just some tweaks to allow for easy debugging.

```
    # disable cache used for testing
    add_header Cache-Control private;
    add_header Last-Modified "";
    sendfile off;
    expires off;
    etag off;
```

Next, lets check out the nginx default site template.

```
vagrant@mgmt:~$ less templates/default-site.j2
```

You will notice the ansible managed comment fact up top here. Then, the only other change, is that we set the server_name, to the hostname of the node serving the request. This is done through an Ansible fact, hence why we did not disable fact gathering in the Playbook. Lets actually jump back to the command line and see how this works. I am

just going to type the command, then we can work through how this works. Ansible web1 -m setup -a filter=ansible_hostname.

```
server_name {{ ansible_hostname }};
```

So, we ran the ad-hoc ansible version, against web1, specified that we wanted to use a module, then provided the setup module. The setup module allows you to gather facts from remote machines, a confusing module name I know, but it is used for gathering facts. Then we provided an argument, saying that we wanted to filter for the ansible hostname fact. This looks through the huge listing of facts returned from the setup module, run against web1, and pulls out the ansible hostname fact. Then down here, you can see that we have the hostname web1.

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

So, if we check out that default site template again, you can see here, that we are going to set the server name to the hostname of our remote machines. This is extremely useful for creating generic templates, then customizing around facts for each machine.

Now that we have chatted about the nginx config, and the default site, lets check out the index.html file. The index.html file will be the one served up to our browser when we connect to each web node through the load balancer.

```
vagrant@mgmt:~$ less templates/index.html.j2
```

You can see it is a pretty simple file, actually my web design skills are lacking, so I did a little reading about how to center text on the screen, there are the links for anyone interested. The interesting bits here, are the E46 heading, then down here, you will see a message about how this request was served by, the web servers hostname, and then its IP address. This is just an added visual verification step, that we are actually sending connections through the load balancer to different web servers.

```
Served by {{ ansible_hostname }} ({{ ansible_eth1.ipv4.address }}).
```

Let me show you how we get this ansible eth1 ipv4 address fact. Lets jump back to the command line, and run our ad-hoc ansbile command against web1 again, but this time change the filter, and set it to ansible eth1. You can see that we are given a bunch of facts about the eth1 device, but you can see the ipv4 heading down here, then the address key, and finally we have the ansible eth1 ipv4 address value. So, if we check out the template again, you should start to see how we get these facts. If you are interested in playing around with them, I suggest just running the ad-hoc ansible command again web1 without a filter, to get a full listing of the facts available.

```
vagrant@mgmt:~$ ansible web1 -m setup -a "filter=ansible_eth1"
web1 | success >> {
    "ansible_facts": {
        "ansible_eth1": {
            "active": true,
            "device": "eth1",
            "ipv4": {
                "address": "10.0.15.21",
                "netmask": "255.255.255.0",
                "network": "10.0.15.0"
            },
            "ipv6": [
                {
                    "address": "fe80::a00:27ff:feba:8b2b",
                    "prefix": "64",
                    "scope": "link"
                }
            ],
            "macaddress": "08:00:27:ba:8b:2b",
            "module": "e1000",
            "mtu": 1500,
            "promisc": false,
            "type": "ether"
        }
    },
    "changed": false
}
```

Okay, so that is three templates down, lets check out the haproxy template before creating our load balanced environment. This is pretty much a stock file, except for a couple bits here and there. The way I captured these files, was to manually install these packages by hand on each machine, then copy these files in their default state back to the management node. This is where I mean that configuration management is not a silver bullet, in that you will have to do things manually at least once, then from there you can start to automate. So, this did take a while to install, test, tweak, etc before I had everything working, then you capture the end result and put that into Ansible. This is one of the things that I love about Vagrant and Ansible, in that I probably destroyed and recreated this environment 25 plus times, while testing for this episode series. This would have been near impossible without automation tools.

```
vagrant@mgmt:~$ less templates/haproxy.cfg.j2
```

Okay, so lets quickly review this file, then get on with the demos. You can see our standard header up here, I highly suggest always adding this. Again, I am not going to talk too much about the standard configuration, as you can easily look that up. The one thing I did change, was to enable a statistics uri. This will allow us to see various statistics about the haproxy, by hitting this special link, and it can be extremely useful for debugging the environment.

```
    # enable stats uri
    stats enable
    stats uri /haproxy?stats
```

Finally, we dynamically build out the backend server list based around the facts provided by Ansible. Do no worry about understanding this too much, as we will review a side by side comparison of what the end result looks like, after we have deployed the environment.

```
backend app
    {% for host in groups['lb'] %}
       listen episode46 {{ hostvars[host]['ansible_eth0']['ipv4']['address'] }}:80
    {% endfor %}
    balance     roundrobin
    {% for host in groups['web'] %}
        server {{ host }} {{ hostvars[host]['ansible_eth1']['ipv4']['address'] }} check port 80
    {% endfor %}
```

But, we are running a for loop, for all hosts in the load balancer group. Then we are putting a line entry down here, that says listen, on the ansible eth0 ipv4 address. The gist of this, is that we want to make sure the load balancer is listening on the correct IP address.

Then we set, the load balancer algorithm to use, in our case using a round robin, which will direct traffic across the nodes, one after another.

Finally, we have a second for loop. This time, we are adding backend web nodes. So, we are saying, for all hosts in the web group, add a server entry here, listing the hostname, then adding its IP address. Again, do not worry about understanding this too much, as we will look at a side by side comparison is just a minute.

Pulling the Trigger (ansible-playbook e46-site.yml)

I think at this point, we have covered everything you need to know, so lets pull the trigger, and setup this load balanced web cluster using Ansible. We can run, ansible-playbook e46-site. I sped this up a little, as it took about a minute and 25 seconds to run.

```
vagrant@mgmt:~$ ansible-playbook e46-site.yml

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

TASK: [install git] ************************************************************
changed: [web3]
changed: [web5]
changed: [web4]
changed: [web1]
changed: [web2]
changed: [web6]
changed: [lb]

PLAY [web] *********************************************************************

GATHERING FACTS ****************************************************************
ok: [web1]
ok: [web4]
ok: [web2]
ok: [web3]
ok: [web5]
ok: [web6]

TASK: [install nginx] **********************************************************
changed: [web2]
changed: [web3]
changed: [web5]
changed: [web1]
changed: [web4]
changed: [web6]

TASK: [write our nginx.conf] ***************************************************
changed: [web2]
changed: [web3]
changed: [web4]
changed: [web1]
changed: [web5]
changed: [web6]

TASK: [write our /etc/nginx/sites-available/default] ***************************
changed: [web2]
changed: [web3]
changed: [web1]
changed: [web4]
changed: [web5]
changed: [web6]

TASK: [deploy website content] *************************************************
changed: [web1]
changed: [web4]
changed: [web3]
changed: [web5]
changed: [web2]
changed: [web6]

NOTIFIED: [restart nginx] ******************************************************
changed: [web1]
changed: [web5]
changed: [web2]
changed: [web3]
changed: [web4]
changed: [web6]

PLAY [lb] **********************************************************************

GATHERING FACTS ****************************************************************
ok: [lb]

TASK: [install haproxy and socat] **********************************************
changed: [lb] => (item=haproxy,socat)

TASK: [enable haproxy] *********************************************************
changed: [lb]

TASK: [deploy haproxy config] **************************************************
changed: [lb]
```

```
NOTIFIED: [restart haproxy] ****************************************************
changed: [lb]

PLAY RECAP ********************************************************************
lb                         : ok=6    changed=5    unreachable=0    failed=0
web1                       : ok=7    changed=6    unreachable=0    failed=0
web2                       : ok=7    changed=6    unreachable=0    failed=0
web3                       : ok=7    changed=6    unreachable=0    failed=0
web4                       : ok=7    changed=6    unreachable=0    failed=0
web5                       : ok=7    changed=6    unreachable=0    failed=0
web6                       : ok=7    changed=6    unreachable=0    failed=0
```

Lets scroll up to where we first ran the e46-site playbook, and work through what is happening. So, the e46-site has three plays, lets see if we can figure out how the output relates to each one. We can see this first block here, it is the play, that is targeting all nodes, where we are install git onto all nodes in our hosts inventory.

Then down here, we have the web play, this is where our web tier gets installed and configured. Since we did not explicitly turn fact gathering off, you can see that facts are gathered about our web nodes, and this is important because we are included these facts in various configure files.

Next, we install nginx, deploy the servers configuration file, setup our default site, push out the index.html file, and finally a restart of nginx is triggered, due to our notification hooks on all of the configuration files.

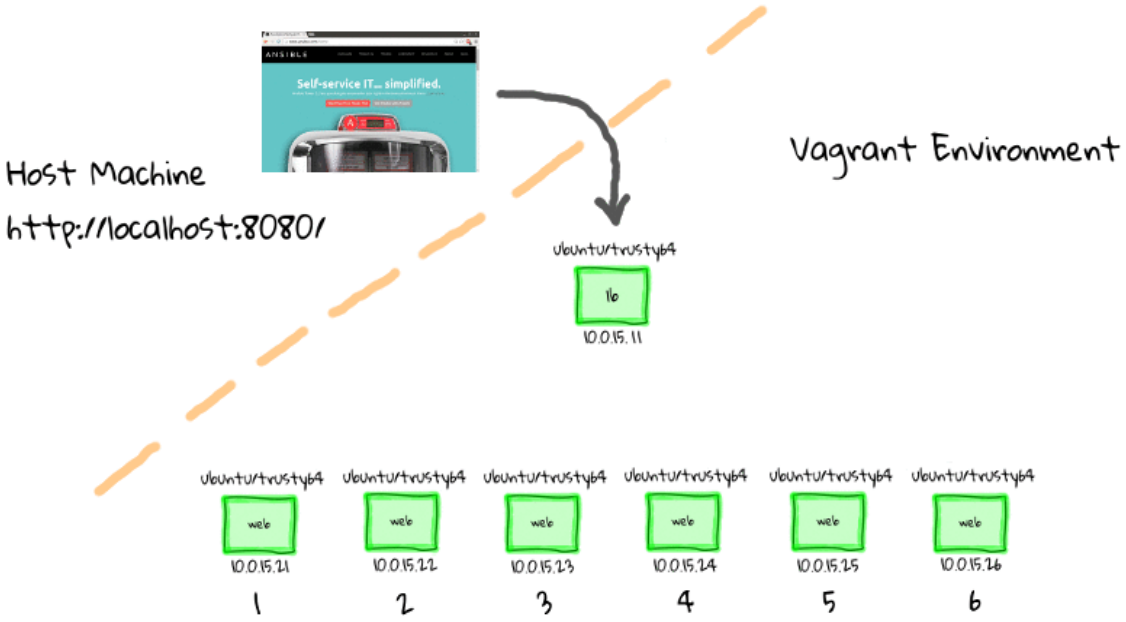
The final play, in the e46-site playbook, configures our load balancer. Again, since we have not turned off gathering facts, this happens by default. Next, we install the required packages, enable haproxy, deploy our custom haproxy configuration file, which is heavily based around facts gathered from our environment. Finally, a restart of haproxy has been triggered since various configuration files have been updated. As always, you will see a playbook recap down here.

Lets just rerun the ansible-playbook e46-site and make sure that nothing changes. Cool, so this acts as a bit of a verification that our environment conforms to what we have defined.

```
vagrant@mgmt:~$ ansible-playbook e46-site.yml
...
PLAY RECAP ********************************************************************
lb                         : ok=5    changed=0    unreachable=0    failed=0
web1                       : ok=6    changed=0    unreachable=0    failed=0
web2                       : ok=6    changed=0    unreachable=0    failed=0
web3                       : ok=6    changed=0    unreachable=0    failed=0
web4                       : ok=6    changed=0    unreachable=0    failed=0
web5                       : ok=6    changed=0    unreachable=0    failed=0
web6                       : ok=6    changed=0    unreachable=0    failed=0
```

So, that is how we configure this environment. Now this might be a little of a let down, as we have had all this build up, and it is as simple as running a single command to build out our environment. I really think this speaks to the power of Vagrant and Ansible. As, you can just download these supporting files and get this up and running rather quickly.

I just wanted to take a moment, and explain how we are going to connect from our host machine, into the Vagrant environment, to the load balancer. Thought the best way, was just to add an overlay and then we can talk about it. So, on this site, we have the machines launched in our Vagrant environment, not shown here is the Ansible management node, but its there too. Then over here, you have your desktop, laptop, or whatever that you are running Vagrant on, but I am going to call it the host machine. In our Vagrantfile, which created this virtual Vagrant environment, we mapped port 8080 from the host machine, into the load balancer on port 80. So, when we connect to localhost 8080 on the host machine, it will be redirected into the Vagrant environment, to the load balancer on port 80.



How about we jump back to the command line, and see how this works, as it will likely make way more sense if you have not used this feature before. The /vagrant mount links back to our project directory on the host machine, so you can see the Vagrantfile in there. Lets just open that up and have a look.

In the second block here, we define how the load balancer should be created, and you can see this forwarded port option here, where we map port 8080 on the host machine, to port 80, on the guest. So, that is how this works. Rather simple, but extremely powerful for testing network services. Lets exit out of here and head back to the diagrams for a minute.

```
vagrant@mgmt:~$ cat /vagrant/Vagrantfile

  # 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"
      # mapping from host 8080 to guest 80
      lb_config.vm.network "forwarded_port", guest: 80, host: 8080
      lb_config.vm.provider "virtualbox" do |vb|
        vb.memory = "256"
      end
  end
```

So, when we make a connection on our host machine to localhost 8080, we are actually being forwarded into the Vagrant environment, then as we hit refresh we will be distributed across our web nodes. Personally, I just find this so cool. As, you can get play around with pretty complex virtual environment, and the overhead for creating them is not really much, once you get the hang of Vagrant and Ansible.

Testing via a Browser

Okay, lets jump over to the browser and test this out in real life. So, we are sitting here on the Ansible website, lets just change the location to localhost 8080, and hit return. Sweet, so we connected into the Vagrant environment, through the load balancer, and web1 served our request. You can see our nice little E46 Demo title, then down here, it says our request was served by web1, and then the IP address. Lets hit refresh a few times and watch this change as we cycle through the web nodes. Pretty cool right.

We can check out the load balancer statistics page too, by using our second browser tab, and navigating too, localhost 8080 haproxy?stats. This is the statistics report generated by haproxy and keeps up to date traffic totals across the load balancer. We can watch these stats change too, by switching back to the first tab, and hitting refresh a couple

times, then coming back to the haproxy report, and hit refresh. You can see that web1, two, and three each served an additional request.

We can also test this out by heading to the command line, and from our host machine, using curl to hit the load balancer. Lets run curl I http://localhost:8080/, and we can just run this a few times. You might remember that backend server header that we added to the nginx configuration file, well we can see that header added here. Our request was served by web4, five, and six. This kind of thing is really useful for debugging your haproxy and web nodes, as you have some insight into who you are talking to.

```
[~]$ curl -I http://localhost:8080/
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 04 Mar 2015 06:03:49 GMT
Content-Type: text/html
Content-Length: 632
X-Backend-Server: web4
Cache-Control: private
Accept-Ranges: bytes

[~]$ curl -I http://localhost:8080/
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 04 Mar 2015 06:03:50 GMT
Content-Type: text/html
Content-Length: 632
X-Backend-Server: web5
Cache-Control: private
Accept-Ranges: bytes

[~]$ curl -I http://localhost:8080/
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 04 Mar 2015 06:03:51 GMT
Content-Type: text/html
Content-Length: 632
X-Backend-Server: web6
Cache-Control: private
Accept-Ranges: bytes
```

Heading back to the haproxy statistics page, we can hit refresh to grab the latest report, and as you can see the request count has been updated. I thought we might use apache benchmark against the load balancer from our host machine too, just for fun. This does not really prove anything, other than padding the stats counts, and just illustrate how a high traffic website might distribute load across machines.

Lets run, ab -n 10000 -c 25 http://localhost:8080/. So we are saying, use apache benchmark, issue 10,000 requests, with a concurrency of 25, and the url is localhost 8080. We can see a running total of completed requests. This goes pretty quickly, but you can see that 10000 requests completed successfully, with zero failures, and you can see the requests per second down here.

```
# installed apache bench on my host machine
sudo apt-get install apache2-utils
```

```
[~]$ ab -n 10000 -c 25 http://localhost:8080/
This is ApacheBench, Version 2.3 (Revision: 655654)
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests


Server Software:        nginx
Server Hostname:        localhost
Server Port:            8080

Document Path:          /
Document Length:        632 bytes

Concurrency Level:      25
Time taken for tests:   15.097 seconds
Complete requests:      10000
Failed requests:        0
Write errors:           0
Total transferred:      8380000 bytes
HTML transferred:       6320000 bytes
Requests per second:    662.37 [#/sec] (mean)
Time per request:       37.743 [ms] (mean)
Time per request:       1.510 [ms] (mean, across all concurrent requests)
Transfer rate:          542.06 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0  17.3      0     999
Processing:    14   37   9.2     36     237
Waiting:       14   37   9.2     36     237
Total:         14   38  19.8     36    1049

Percentage of the requests served within a certain time (ms)
  50%     36
  66%     40
  75%     42
  80%     44
  90%     50
  95%     54
  98%     60
  99%     64
 100%   1049 (longest request)
```

Heading back to the haproxy stats page, we can hit refresh again, and now you can see that we have some meaningful numbers in here. So, this is how you could use Ansible to stand up a load balanced web cluster.

I just wanted to chat about a couple more things before we end this episode. Lets have a look at what the haproxy servers configuration file looks like, as this was heavily generated using a template, with facts from our environment. I have opened up two terminal windows here, both connected to the management node, and we are going to look at a side by side comparison of what the haproxy configuration file. So, in the top window, lets cat the haproxy template, from the management node, just to get an idea of what these for loops look like.

```
backend app
    {% for host in groups['lb'] %}
       listen episode46 {{ hostvars[host]['ansible_eth0']['ipv4']['address'] }}:80
    {% endfor %}
    balance     roundrobin
    {% for host in groups['web'] %}
        server {{ host }} {{ hostvars[host]['ansible_eth1']['ipv4']['address'] }} check port 80
    {% endfor %}
```

Then in the bottom terminal here, lets ssh from the management node, into the load balancer. What is so cool about this, is that this is actually our first time logging into this machine, and it was fully configured without us ever logging into it manually. From here, lets cat the /etc/haproxy/haproxy.cfg file, and this last block here is what we are looking for. I just thought this would be a good way of showing you what the before and after looks like for this template. You might actually find it useful to review these files on your own, or play around with various facts, as it is a great way to learn.

```
backend app
           listen episode46 10.0.2.15:80
        balance     roundrobin
            server web1 10.0.15.21 check port 80
            server web2 10.0.15.22 check port 80
            server web3 10.0.15.23 check port 80
            server web4 10.0.15.24 check port 80
            server web5 10.0.15.25 check port 80
            server web6 10.0.15.26 check port 80
```

Hopefully at this point, you have a pretty good idea of what a real world Ansible use case would be. I actually had several additional demos planned, by way of these four playbooks, but when working through the editing process for this episode, I realized it was already running way too long. So, what I will do, is leave this as homework if you are interested, then you can work through these as you have time.

Let me just show you what this e46-role-site playbook looks like. As you can see, it has three plays in it, common, web, and lb, just like the e46-site playbook we looked at earlier. There are a couple interesting differences though, it is much sorter, there is no tasks sections, but rather this roles definition.

```
vagrant@mgmt:~$ cat e46-role-site.yml
---

# common
- hosts: all
  sudo: yes
  gather_facts: no

  roles:
    - common

# web
- hosts: web
  sudo: yes

  roles:
    - web

# lb
- hosts: lb
  sudo: yes

  roles:
    - lb
```

Roles allow you to move the configuration out of the playbook and into a file and directory structure, called a role. Let me show you what I mean, if we list the directories again, you will see this roles folder. We can run tree against it, and get an expanded view of the files and directories under it. So, here you can see that we have a web role defined, along with tasks, handlers, and templates directories. Then in those folders we have configuration files and our templates files.

```
vagrant@mgmt:~$ tree roles/
```

Roles allow you to remove bulky configuration out of the playbook, and into a folder structure called a role. The entire goal of this, is to make things modular, so that you can share and reuse roles, along with not having playbooks that are thousands of lines long. Feel free to play around with these files, see what they do, you can just run ansible playbook again then, just like regular playbooks. I have actually linked to the Roles documentation page too, you can find that in the episode nodes below, and it goes into much more detail and I have done here.