Episode #47 - Zero-downtime Deployments with Ansible (Part 4/4)

About Episode - Duration: 24 minutes, Published: Apr 02, 2015

In this episode, we will do a zero-downtime rolling website deployment, across a cluster of nginx web nodes, sitting behind a haproxy load balancer. We will use Ansible to orchestrate the deployment in a repeatable and consistent way.

Download: mp4 (https://s3.amazonaws.com/sysadmincasts.com/static/videos/47-zero-downtime-deployments-with-ansible-part-4-4.mp4) or webm (https://s3.amazonaws.com/sysadmincasts.com/static/videos/47-zero-downtime-deployments-with-ansible-part-4-4.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 #46 - Configuration Management with Ansible (Part 3/4) (/episodes/46-configuration-management-with-ansible-part-3-4)
- Ansible Documentation (http://docs.ansible.com/index.html)
- Ansible - Configuration File (forks limit) (http://docs.ansible.com/intro_configuration.html#forks)
- Ansible - Rolling Update Batch Size (serial keyword) (http://docs.ansible.com/playbooks_delegation.html#rolling-update-batch-size)
- Ansible - Continuous Delivery and Rolling Upgrades (pre_tasks and post_tasks) (http://docs.ansible.com/guide_rolling_upgrade.html)
- Github: jweissig/episode-47 (website release-0.01, 0.02, and 0.03) (https://github.com/jweissig/episode-47)
- Bootstrap: HTML, CSS, and JS framework (http://getbootstrap.com/)

- Github: Ansible Examples (https://github.com/ansible/ansible-examples)
- Ansible Galaxy (https://galaxy.ansible.com/)
- Fedora Project - Ansible Repo (https://infrastructure.fedoraproject.org/cgit/ansible.git)

In this episode, we will do a zero-downtime rolling website deployment, across a cluster of nginx web nodes, sitting behind a haproxy load balancer. We will use Ansible to orchestrate the deployment in a repeatable and consistent way.

I think this will be a fitting conclusion to our look at Ansible, as it not only mimics what you might do in real life, but also leverages what we have learned throughout this episode series.

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, where we used both ad-hoc commands and playbooks, to deploy packages, configuration files, and restart services. Then in part three, episode #46, is where we built out the Vagrant environment even more, to support a load balanced web cluster. Finally, in part four, this episode, we are going to close out the series, by trying our hand at multiple zero-downtime rolling website deployment across our cluster of web nodes.

Defining the Problem

We are going to pick up from where we left off in part three, that being we have a Vagrant environment outfitted with a haproxy load balancer, backed by six nginx web servers. Then we used a web browser, to connect into the the Vagrant environment, and watched as or requests went through the load balancer and were serviced by our cluster of web servers.

So, what is the point of this episode, what problem are we trying to solve, and what does a zero-downtime rolling software deployment actually mean? Well, for the sake of the examples in this episode, it boils down to continually serving live traffic, while updating the web nodes one by one, all without having any user noticeable downtime or dropped connections.

Modelling a Zero-Downtime Rolling Deployment

So, with that in mind, lets model how we might go about doing one of these rolling deployments manually. Lets jump over to an editor, so we can mock out how this is going to work at a high level, and you can see I already have some basic tasks here. For each node that we update, we want to verify that nginx is installed, and it has the correct config file. These steps are not required for a deployment, but is makes sense to ensure our hosting environment is in a sane state, and that is conforms to our known good requirements. Then, we will remove the existing website, and deploy our new website, and finally restart nginx if needed.

Now, if we just did something like this, users would notice downtime because we have not removed live traffic going to these machines via the load balancer. So, we need some type of pre-task, where we tell the load balancer to put this web server into maintenance mode, so that we can temporarily disable traffic from going to it, as it gets upgraded. Lets add a block up here, that says a pre-task will be to disable web node in haproxy.

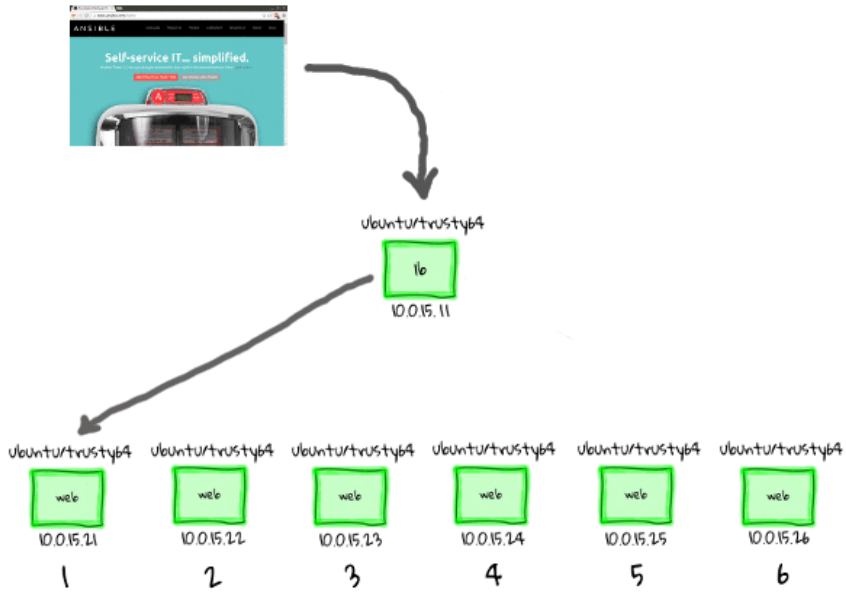
So, this is our pre-task, where we disable traffic, then down here, we upgrade the node using these various tasks. Finally, we need some type of post-task, which will enable traffic to this web node again, by taking it out of maintenance mode. So, this is the rolling zero-downtime pattern at a high level, we just use a pre-task to disable traffic, do the upgrade while no requests are being serves, and finally enable traffic again. Finally, we just take this pattern, and loop over all of our web servers on by one, and you have fancy rolling zero-downtime deployment.

```
pre_tasks:
  - disable web node in haproxy

tasks:
- install nginx package (package)
- write nginx.conf config file (file)
- clean existing website content
- deploy website content (from github)
- start nginx (service)

post_tasks:
  - enable web node in haproxy
```

I have actually had to do this a few times without using an orchestration tool, and as you can imagine, there are tons of little tasks that need to be done, and it is extremely easy to miss something, then you have unknown configuration errors popping up all over the place because of a missed step. This turns into a total nightmare, and you do not want

to do frequent deployments, as it is so painful, and you run a high risk of blowing something up along the way, just because it is so tedious.

This is where Ansible comes to the rescue, in that we can use it as an orchestration tool to manage these complex zero-downtime rolling software deployment for us. I thought it might make sense to model how a rolling deployment works via some diagrams, because even though it is a pretty simple at heart, there is some complex task orchestration happening.



Picking Up Where We Left Off

Okay, so now that we know what the problem looks like, and how we want to solve it, lets jump to the command line and make it happen.

As I said a couple minutes ago, we are going to pick up from where we left off in part three of this series. Lets just change into the e45 project directory, you can see that we have our Vagrantfile, bootstrap script, and examples directory. We can check the environment state by running, vagrant status, and we still have our eight virtual machines up and running. If none of this looks familiar, make sure you watch parts two and three, as we dive pretty deep into how this gets created.

```
[~]$ cd e45
[e45]$

[e45]$ ls -l
total 16
-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 15:03 Vagrantfile

[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)
```

Lets log into the management node, by running vagrant ssh mgmt, kind of a good by product of learning about Ansible in this virtual environment, is that you get lots of practical experience with Vagrant too.

```
[e45]$ vagrant ssh mgmt
```

The majority of these supporting files should look familiar, as we already worked through the bulk of them, but today we are going to focus on these e47 playbooks, which will be used to flush out how rolling deployments work under the hood.

```
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  103 Mar  6 10:13 e47-parallel.yml
-rw-rw-r-- 1 vagrant vagrant 1963 Mar  6 14:18 e47-rolling.yml
-rw-rw-r-- 1 vagrant vagrant  115 Mar  6 10:13 e47-serial.yml
drwxrwxr-x 2 vagrant vagrant 4096 Jan 31 05:03 files
-rw-r--r-- 1 vagrant vagrant   63 Mar 31 08:30 inventory.ini
drwxrwxr-x 5 vagrant vagrant 4096 Feb  9 19:26 roles
drwxrwxr-x 2 vagrant vagrant 4096 Feb  6 20:07 templates
```

Just to make sure we are both on the same page, I thought it might make sense to run the e46-site playbook again, this will configure our load balanced web cluster to a known good state. My thinking is that, you might have been playing around with the environment in part three, so I wanted to just make sure that we are working from the same starting point. So lets run, ansible-playbook e46-site.

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

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

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

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

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

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

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

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

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

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

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

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

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

TASK: [deploy haproxy config] **************************************************
ok: [lb]

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
```

We can verify this is all up and running, by opening up a web browser, and connecting to localhost 8080. You can see the example website here from episode 46, and if we hit refresh a few times, you can see that we cycle through the web nodes. Lets also open up a second tab here, and connect into the haproxy load balancers statistics page, you can see our six web nodes here, and they have served one request each. We went through that pretty quickly, but I just wanted to make sure we are on the same page, so that you can reproduce my results if you are following along.

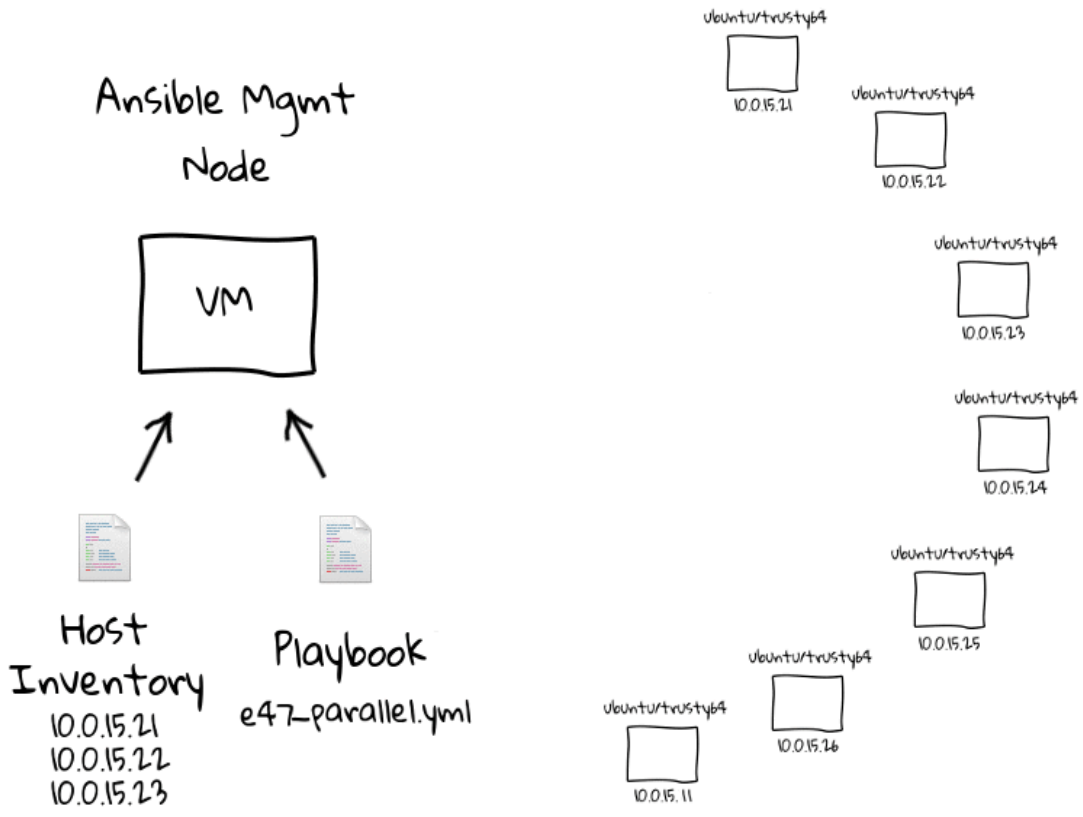
Parallel Task Execution

One of the key things that makes a rolling deployment possible, is running tasks in a serial workflow, rather than a parallel workflow. Up until this point, all of our examples through the series have focused on running things in parallel. So, before we look at doing website deployments with Ansible, I want to make sure the differences between parallel and serial task execution are crystal clear.

We have two example playbooks here, the e47-parallel playbook, along with the e47-serial playbook. Lets just cat the e47-parallel playbook, and you should recognize pretty much everything in here, as we have heavily covered these basic concepts. We are targeting all hosts, turned fact gathering off, then down here, we have one task defined, and it will connect to the remote host, use the shell module, and sleep for 5 seconds.

```
vagrant@mgmt:~$ cat e47-parallel.yml
---
- hosts: all
  gather_facts: no

  tasks:

  - name: sleep for 5 seconds
    shell: /bin/sleep 5
```

Probably the best way to describe the differences between parallel and serial task execution, is through the use of a few diagrams, as I really like to have a visual sense of what is happening behind the scenes. This is a super simple concept, but the implication is pretty big, especially when you want to do rolling updates. This diagram should look familiar, as I have heavily used these through the series, so when we run this e47-parallel playbook from the management node, we are going to connect to all of these nodes in parallel, and sleep for 5 seconds, before returning.



Now that you know what it looks like, lets head to the command line, and run it against our client nodes to see it in action. Lets run, ansible-playbook e47-parallel, actually lets prefix this with the time command, just so that we get a sense for how long this takes to complete.

```
vagrant@mgmt:~$ time ansible-playbook e47-parallel.yml

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

TASK: [sleep for 5 seconds] ****************************************************
changed: [web2]
changed: [web5]
changed: [web3]
changed: [web4]
changed: [web1]
changed: [web6]
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
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


real  0m13.413s
user  0m0.371s
sys 0m0.229s
```

What is interesting about this is that is takes around 13 seconds to complete, even though we are connecting to these machines in parallel, and only sleeping for 5 seconds. So, this brings up an interesting Ansible default. So, lets jump over to the Ansible documentation site for a minute, and you will find contained within the configuration file options, this folks define. This fork option acts as a rate limiter for the number of parallel jobs that we can spawn at a time, and it just so happens to be set extremely low, at 5. So, we actually bumped up against this limit while doing our testing, and it was noticeable because our timed task should have only taken about 5 seconds, but was really taking 10 second, and the reason is that we hit the folk limit.

```
vagrant@mgmt:~$ time ansible-playbook e47-parallel.yml

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

TASK: [sleep for 5 seconds] ****************************************************
changed: [web2]
changed: [web1]
changed: [web4]
changed: [web6]
changed: [lb]
changed: [web3]
changed: [web5]

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
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


real  0m5.641s
user  0m0.238s
sys 0m0.236s
```

Lets bump this extremely conservative rate limit up to more reasonable value, by running vi against our local ansible.cfg, and just add a line down here, forks equals 15, this will allow us to run 15 parallel jobs at once. Even this number is conservative, but it will work for our examples, you might want to pick something higher.

Okay, so with that out of the way, lets rerun ansible-playbook e47-parallel, and lets not forget to prefix this with the time command. It seemed to go a little quicker, we are now below 10 second times, and this includes some ssh connection overhead too. Lets rerun again, and this time we are down to about 5 and a half seconds, much better. So, that is parallel task execution in a nutshell.

We kind of went off on a tangent here, but like I mentioned, you will almost certainly run into this issue if you have more than 5 machines, so it is work spending a couple minutes on it.

Serial Task Execution

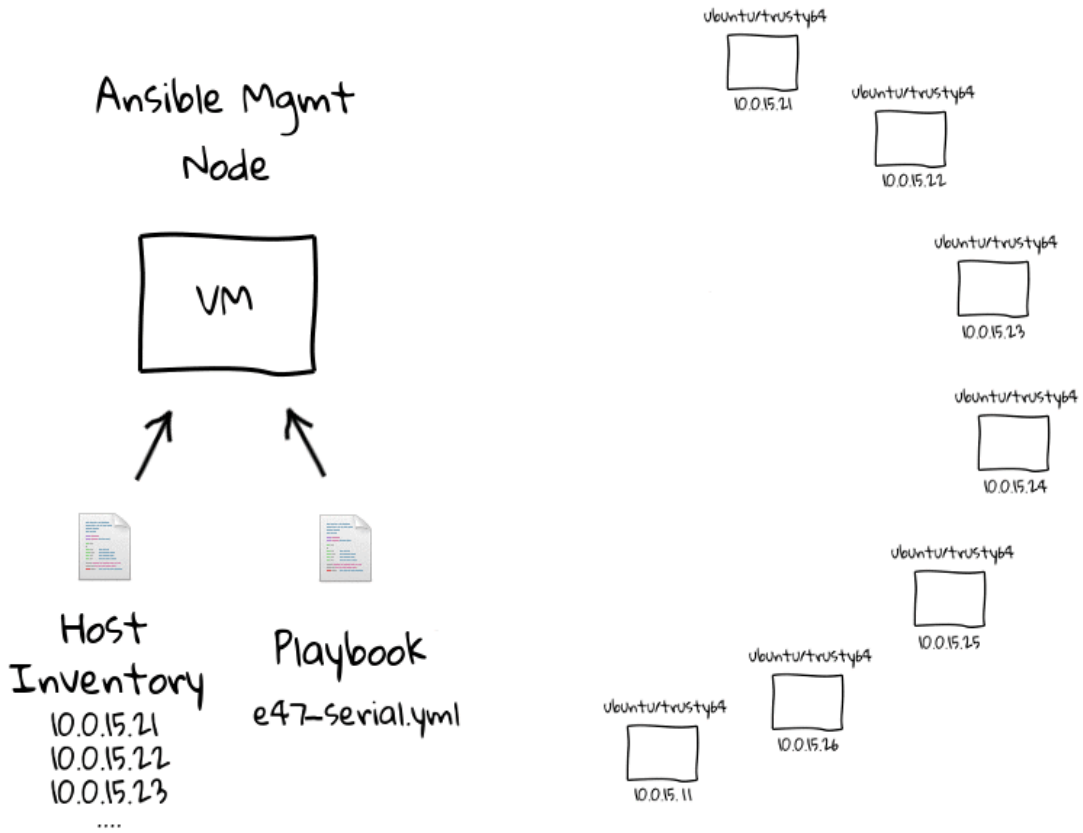
One of the key pieces of the rolling update puzzle, is the ability to run tasks in a serial manner, one after another, rather than in parallel. To learn more about this, we can jump over to the delegation, rolling updates, and local actions documentation page. This page is packed with useful guidance for doing rolling updates, but we are just going to look at this rolling update batch size heading here, and we can see this playbook serial keyword option. This allows us to define the number of machines that we want to connect to at a time, rather than everything in parallel, we can narrow this down to one at a time, several at a time, or percentages, like 30% of machines at a time for example.

This will likely make much more sense if we just look at an example of this in action. So, lets hop back to the command line for a minute. I am just going to cat the e47-parallel playbook, so that we have something to compare too, then lets cat the e47-serial playbook. The only difference between these two playbooks, is this single serial option, but it has a dramatic effect on how this are run.

```
vagrant@mgmt:~$ cat e47-parallel.yml
---
- hosts: all
  gather_facts: no

  tasks:

  - name: sleep for 5 seconds
    shell: /bin/sleep 5
```

```
vagrant@mgmt:~$ cat e47-serial.yml
---
- hosts: all
  gather_facts: no
  serial: 1

  tasks:

  - name: sleep for 5 seconds
    shell: /bin/sleep 5
```

Here are a few diagrams which model how this works behind the scenes. When we run the e47-serial playbook on the management node, it will first establish a connection to web1, run our sleep command, and when it is done, move onto the next node, when web2 is done, it moves onto web3, after web3, we do web4, and so on.



Lets jump back to the command line and see this in action. We can run ansible-playbook e47-serial, oh and lets not forget about prefixing this with time, just so that we can get some stats. You can already see that this looks much different, in that we are running one task at a time, compared to doing this in parallel, which is the default option. I am just going to speed this up a little, as we do not need to wait around for this, and you can see the time difference is pretty huge. We can just rerun this for dramatic effect, again I have sped it up, and the final result is pretty obvious to see. The e47-parallel playbook was getting pretty close to 5 seconds, where as the e47-serial playbook clocks in around 36 seconds.

```
vagrant@mgmt:~$ time ansible-playbook e47-serial.yml

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

TASK: [sleep for 5 seconds] ****************************************************
changed: [web1]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web2]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web3]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web4]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web5]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web6]

TASK: [sleep for 5 seconds] ****************************************************
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
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


real  0m45.497s
user  0m0.351s
sys 0m0.223s


vagrant@mgmt:~$ time ansible-playbook e47-serial.yml

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

TASK: [sleep for 5 seconds] ****************************************************
changed: [web1]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web2]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web3]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web4]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web5]

TASK: [sleep for 5 seconds] ****************************************************
changed: [web6]

TASK: [sleep for 5 seconds] ****************************************************
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
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


real  0m36.034s
```

```
user  0m0.263s
sys 0m0.151s
```

So, even though the end result is similar, in that we connected to each node, and slept for 5 seconds, task execution was orchestrated much differently, due to the serial option being set.

Pre and Post Tasks

Lets take a look at that text file again, where we mocked up how a rolling deployment would actually happen. So, we have this main task section pretty much figured out, in that we know how to do rolling upgrades using the serial option. But, what about these pre and post tasks, where we turn off traffic going into the web node before the upgrade, then turn on traffic again after the upgrade is complete?

Well, there is an Ansible documentation page on that too, called continuous delivery and rolling upgrades. This page is actually where I got the idea for this episode, and they go into a crazy amount of detail for making these rolling deployments a reality, so if you are interested in this type of stuff, I would highly recommend reading this page too.

If we scroll about three quarters of the way down this page, there is a code block, which talks about how we can define these pre tasks. This is actually baked right into the Ansible playbook logic, so we do not need any special hacks, we just define this pre tasks section, then a little farther down this page, post tasks are also defined. So, we can just utilize these playbook options, and we should be able to do exactly what we want to. In that we have our pre task to disable traffic, do the update steps, then enable traffic again.

E47-Rolling Zero-Downtime Playbook

Lets head back to the command line and actually see what this e47-rolling zero-downtime deployment playbook looks like. I think we have most of the problems figured out now, we know the difference between parallel and serial task execution, we know how pre and post task work, so lets open up the e47-rolling playbook and see what the final product looks like.

```
vagrant@mgmt:~$ less e47-rolling.yml
```

If you were going to do this in real life, you might want to break this playbook into roles, just in minimize the amount of text in it, but for the sake of having everything in one place to review, I have left this as one large monolithic file. By the way, this is actually just the e46-site playbook, adapted for rolling updates, so most of the bits should look familiar.

The first play here, targets all nodes, and installs git, which will be used to pull down our website content from a repository I have setup for this episode. Next we have the web play, but I will come back to this in a second, as we will be spending most of our time reviewing this. The last play here, targets our load balancer, and this is the exact same setup used in the e46-site playbook, so you should already understand what is happening in here.

Now that you have a general idea of the main blocks of code in here, lets focus our attention on the web play, as that is where all the interesting bits for the rolling deployment are located.

We have most of the standard bits in here, we are targeting all web nodes, using sudo to run commands, then down here we have defined a custom variable or fact, which specifics the website version we want to deploy, but I will talk more about in a minute, and finally, we have specified that we want to executing things in a serial manner, one node at a time.

Here, we have our pre-task block, so for each node we update today, this pre-task block will go first. Lets work through what is happening here, we define a task name, then we define the task action, here we are using the shell module to echo this string of text, piping it over to socket cat, which lets the load balancer know we want put a node into maintenance mode.

This is pretty specific to haproxy, but the load balancer can be configured to listen on a local socket, then we can pass the load balancer commands, just by echoing things into it. So, we are sending the disable server command, and specifying which server we want to disable based off an Ansible gathered fact.

Finally, we have this delegate to keyword, this allows us to specify a node we want to run this command on, and since we are in the web play here, we need to delegate this command over to a load balancer, and that is what this with items line here does. So, we are saying, any nodes in the load balancer group, delegate this command to them.

It make sense to go and run these pre and post commands manually on the load balancer, after we review this playbook, just so that we can see what this looks like outside of automation tools.

Next, there is a pretty basic tasks section here, we make sure the nginx package is installed, deploy our nginx server and default site configuration files, chances are this will already be done, but does not hurt to make sure everything is consistent.

This next option is a little extreme, but I just wanted to show you that we are totally blowing away the website content, then down here we populate the website content via a git pull from github. I thought it might be useful to mimic pulling a website out of version control, and you can see here we are even specifying the version we want to deploy based off a custom variable we defined earlier. If we scroll up here, you can see we are setting the app_version to release-0.01, and I have tagged several example website release versions on my public Github page.

Lets check out this link, and I can walk you through how it works, in the repository we have a readme, an index.html file, and a css stylesheet. So, when the playbook website deployment task is executed, it will use the git module to pull this website content from my repository. If we click on this releases heading, you can see three releases here, release-0.01, 0.02, and 0.03. These are the different websites that we are going to use for our rolling deployment tests.

So, that is how the website deployment bit works, lets jump back to the playbook and see how the post-task works. This is basically just the pre-task, but in reverse, as you would expect, since we marked the node off-line, via the pre-task, updated the website using the main tasks block, and finally down here, we re-enable the node. As you can see, there is nothing too crazy in here, and this actually closely mimics what you might do manually, but just in a format that Ansible can orchestrate for us.

HAProxy Maintenance Mode

So, with the playbook out of the way, lets investigate how maintenance mode with with haproxy. If we hop over to the load balancers statistics page, you can see that all six web nodes are in an active and up state, based off their colour status code. Lets log into the load balancer manually, and disable a couple of these, using the commands from our playbook, as this will hopefully give us a pretty good feel for how it all comes together.

So, lets just ssh from the management node, into the lb machine. From here, we can sudo to root, by running sudo su -. Then, I am just going to copy the command from the playbook, we were using an Ansible fact here, but lets just assume we wanted to mark web1 off-line. Oh yeah, this episode46 bit here, comes from the haproxy configuration files, from where we defined our back end web nodes, and you can also see this via the statistics page here.

```
vagrant@mgmt:~$ ssh lb
vagrant@lb:~$ sudo su -
```

Okay, so as you can see here, web1 has been put into a down for maintenance state, and this means that no traffic will be sent its way. Lets disable web2 too, just to see what that looks like, and if we fresh the page here, you can see that both web1 and web2 are down for maintenance. So, our rolling deployment playbook, will mark a node off-line, one by one, update the website content, then mark them back on-line. Since we are doing these one at a time, we will always have available nodes serving traffic, so there will be no noticeable downtime from the users perspective.

```
root@lb:~# echo "disable server episode46/web1" | socat stdio /var/lib/haproxy/stats
root@lb:~# echo "disable server episode46/web2" | socat stdio /var/lib/haproxy/stats
```

Lets enable these nodes again, by running the enable server command, for both web1 and web2, and then we can verify that it actually worked by checking the statistics interface again.

```
root@lb:~# echo "enable server episode46/web1" | socat stdio /var/lib/haproxy/stats
root@lb:~# echo "enable server episode46/web2" | socat stdio /var/lib/haproxy/stats
```

Running ansible-playbook e47-rolling

Okay, at this point, I think we have a handle on everything, and we have learned a lot about what the work flow looks like for a rolling deployment. So, lets run this zero-downtime website deployment, by typing, ansible-playbook e47-rolling.

```
vagrant@mgmt:~$ ansible-playbook e47-rolling.yml

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

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

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

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

TASK: [disable server in haproxy] **********************************************
changed: [web1 -> lb] => (item=lb)

TASK: [install nginx] **********************************************************
ok: [web1]

TASK: [write our nginx.conf] ***************************************************
ok: [web1]

TASK: [write our /etc/nginx/sites-available/default] ***************************
ok: [web1]

TASK: [clean existing website content] *****************************************
changed: [web1]

TASK: [deploy website content] *************************************************
changed: [web1]

TASK: [enable server in haproxy] ***********************************************
changed: [web1 -> lb] => (item=lb)

TASK: [disable server in haproxy] **********************************************
changed: [web2 -> lb] => (item=lb)

TASK: [install nginx] **********************************************************
ok: [web2]

TASK: [write our nginx.conf] ***************************************************
ok: [web2]

TASK: [write our /etc/nginx/sites-available/default] ***************************
ok: [web2]

TASK: [clean existing website content] *****************************************
changed: [web2]

TASK: [deploy website content] *************************************************
changed: [web2]

TASK: [enable server in haproxy] ***********************************************
changed: [web2 -> lb] => (item=lb)

TASK: [disable server in haproxy] **********************************************
changed: [web3 -> lb] => (item=lb)

TASK: [install nginx] **********************************************************
ok: [web3]

TASK: [write our nginx.conf] ***************************************************
ok: [web3]

TASK: [write our /etc/nginx/sites-available/default] ***************************
ok: [web3]
```

```
TASK: [clean existing website content] *********************************
changed: [web3]

TASK: [deploy website content] *****************************************
changed: [web3]

TASK: [enable server in haproxy] ***************************************
changed: [web3 -> lb] => (item=lb)

TASK: [disable server in haproxy] **************************************
changed: [web4 -> lb] => (item=lb)

TASK: [install nginx] **************************************************
ok: [web4]

TASK: [write our nginx.conf] *******************************************
ok: [web4]

TASK: [write our /etc/nginx/sites-available/default] *******************
ok: [web4]

TASK: [clean existing website content] *********************************
changed: [web4]

TASK: [deploy website content] *****************************************
changed: [web4]

TASK: [enable server in haproxy] ***************************************
changed: [web4 -> lb] => (item=lb)

TASK: [disable server in haproxy] **************************************
changed: [web5 -> lb] => (item=lb)

TASK: [install nginx] **************************************************
ok: [web5]

TASK: [write our nginx.conf] *******************************************
ok: [web5]

TASK: [write our /etc/nginx/sites-available/default] *******************
ok: [web5]

TASK: [clean existing website content] *********************************
changed: [web5]

TASK: [deploy website content] *****************************************
changed: [web5]

TASK: [enable server in haproxy] ***************************************
changed: [web5 -> lb] => (item=lb)

TASK: [disable server in haproxy] **************************************
changed: [web6 -> lb] => (item=lb)

TASK: [install nginx] **************************************************
ok: [web6]

TASK: [write our nginx.conf] *******************************************
ok: [web6]

TASK: [write our /etc/nginx/sites-available/default] *******************
ok: [web6]

TASK: [clean existing website content] *********************************
changed: [web6]

TASK: [deploy website content] *****************************************
changed: [web6]

TASK: [enable server in haproxy] ***************************************
changed: [web6 -> lb] => (item=lb)

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

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

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

TASK: [Enable HAProxy] ********************************************************
ok: [lb]

TASK: [Configure the haproxy cnf file with hosts] *****************************
ok: [lb]

PLAY RECAP ********************************************************************
lb                         : ok=5    changed=0    unreachable=0    failed=0
web1                       : ok=9    changed=4    unreachable=0    failed=0
web2                       : ok=9    changed=4    unreachable=0    failed=0
web3                       : ok=9    changed=4    unreachable=0    failed=0
web4                       : ok=9    changed=4    unreachable=0    failed=0
web5                       : ok=9    changed=4    unreachable=0    failed=0
web6                       : ok=9    changed=4    unreachable=0    failed=0
```

What I find so cool about this, is that it does a complete rolling zero-downtime deployment in about 40 seconds on my system, and I would think that you could easily spend an hour or more doing this manually.

Lets scroll up here, and work through what tasks were executed on web1 for example, it will be the same on all nodes, but we can get a pretty good idea of the overall tasks being preformed.

First, we disable the web server in haproxy, so that no new traffic will be directed its way. Next, we make sure the web server is installed, has the correct server configuration file, along with default site file. Then we delete the existing website, and have our website deploy step connect out to Github, where it pulls down the release version we defined. Finally, we enable the web server in haproxy, so that it can handle connections again. Then is moves onto web2. These simple scripts will same you many many hours of manual work if you need to do something similar. The practical applications goes far beyond website updates too, maybe you could do rolling patch deployments, or rolling application upgrades, the sky is really the limit once you know these tools exist.

Lets jump over to our web browser and have a look at the web site. You can see up here, we have our E47 Version 1 website, and I pulled this example template off the bootstrap frameworks website. As always, I have linked this site in the episode nodes below, and if you ever need to create a half decent looking website, but have limited design skills, then make sure you check out the bootstrap framework, as it is a fantastic resource. We can also check out the load balancers statistics page here too, just to verify that everything is in an on-line and active state.

Okay, so I thought it would be kind of cool to do a deployment, while hitting refresh on the load balancers statistics page, so that we can watch Ansible marking nodes off-line, and then back on-line, as it does the rolling update. Not really any practical reason for this, just that it gives you a sense of how this tools works in the background, so lets jump back to the command line and make that happen.

Lets open up the e47-rolling playbook in an editor and adjust the app release version from 0.01 to 0.02. This will allow us to deploy the second revision of our fancy website design.

We actually need to make one more tweak, because I made a mistake using rm to delete our website content, since it does not clean up the files correctly as we git deploy over top. This is a bit of a pain, since I have already released the supporting material for part two and three of this series, so we need to do a bit of a live patch here. I have updated the supporting materials download too, so you might have this already fix, but if not, we are going to replace the rm command, with an Ansible file modules which delete the content for us. Sorry about this.

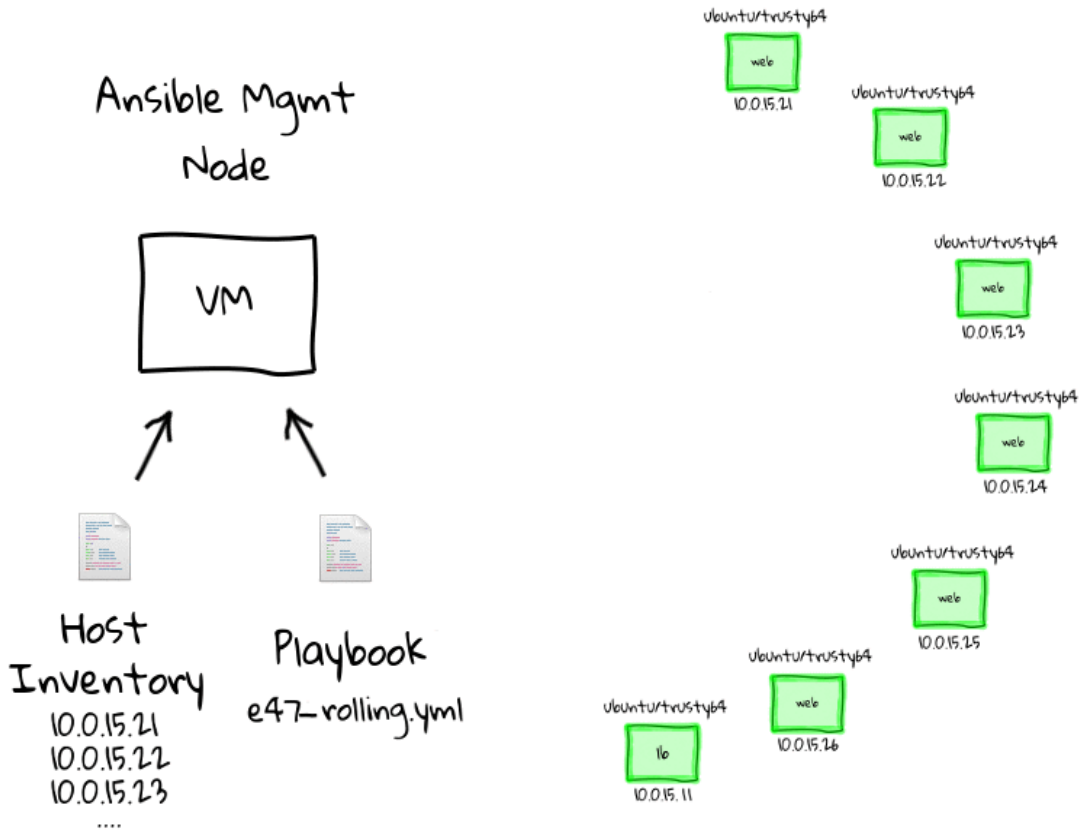
```
  - name: clean existing website content
    #shell: rm -f /usr/share/nginx/html/*
    file: path=/usr/share/nginx/html/ state=absent
```

Okay, so lets run ansible-playbook e47-rolling again, then quickly pop over to our haproxy statistics page, and start hitting refresh.

```
vagrant@mgmt:~$ ansible-playbook e47-rolling.yml
```

Pretty cool right, web1 goes down for maintenance, comes back, then web2 goes down, comes back, and so on. Just something about this that I find so cool. Not only do these deployments save you tons of time, but we reduce the human error factor greatly, as you are not do each of these steps via some checklist. Okay, so lets switch over to the website tab here, and hit fresh a few time, and you can see out latest design change.

Well, I just wanted to show you a final diagram, and we will close out this series. You might be thinking, hey, there is a third website release that can still be deployed, well, I will leave that as homework if you are interested.



For me, these demos and diagrams speak to the power of using Ansible to manage change, both for configuration management, and as an orchestration tool.

So, where can you go from here? Well, you can find some additional links in the episode notes below, but I would recommend checking out the Ansible example playbooks hosted on Github, as they can be pretty fun to work though.

Also, check out Ansible Galaxy, where you can explore playbooks created by the community, to do all types of things. This is a fantastic resource, which will likely save you tons of time. I also recommend checking out Github, as it is a pretty good resource, and I just use the code search option, and type in Ansible.

Finally, if you decide to go crazy with Ansible, I would highly suggest checking out the Ansible playbook repository used by the Fedora Project. You can download their production playbooks, and get a feeling for what a real live environment looks like. Lots can be learned from reviewing how they structure their code, how they abstract things, and it gives you a sense for what is really possible with Ansible.