Ansible Configuration Management Boot Camp

Lab Guide

63100LG_5.0_2018

Ansible Configuration Management Boot Camp

Lab Guide

Prerequisite: Connecting to Your Ansible Control Machine

- Your lab environment is a series of Docker containers running on a virtual machine. This means that we need to attach to our Ansible control container before we get started. To do this follow these steps.

- Connect to the virtual machine using SSH. Use your terminal if your workstation is a Linux machine or a Mac. If your workstation is a Windows machine, you will need an application such as PuTTY to connect. If you need PuTTY download it now. Your instructor will provide you with the credentials needed to connect.

```
1
2  ssh usernameprovided@hostnameprovided
3
```

- Once connected to virtual machine via SSH we will need to setup our environment. To do so type the following into the terminal.

```
1
2  sudo apt-get install git
3
```

- After **git** is installed, use the following command to obtain the environment files.

```
1
2  git clone https://github.com/mattdavis0351/AnsibleBootCamp.git
3
```

- Navigate to the cloned environment files.

```
1
2  cd AnsibleBootCamp/Ansible_class
3
```

- Next you will need to execute **setup.sh** to start the environment. Type the following.

```
1
2  sudo ./setup.sh
3
```

- The output of the above command should look like the screen shot below.

```
CONTAINER ID    IMAGE                                COMMAND
fda1cabd099b    ansibleclass_ansiblecontroller       "/run.sh"
42649f76698a    ansibleclass_centos7                 "/bin/sh -c 'service..."
ad457c3eea8a    ansibleclass_ubuntu                  "/run.sh"
```

- Using the CONTAINER ID of the image name **ansibleclass_ansiblecontroller** we will use the following command to attach to that container, from there we will conduct the remainder of these labs.

```
sudo docker exec -it fda /bin/bash
```

- Notice how we only needed the first three characters of the CONTAINER ID to attach to the desired container. If the first three characters are unique this works, if they weren't unique we would need to include more characters.

- If your prompt has changed to the following example, then you know you have successfully attached to the Ansible control container. **Note that your CONTAINER ID will mostly likely be different from what you see in the lab guide.**

```
root@fda1cabd099b:
```

- Reminder: The above steps do not have anything to do with Ansible and are strictly for connecting to our lab environment for this course! Your actual work environment might require different connection steps prior to using Ansible.

Exercise 1: Installing Ansible

- Before installing ansible a few prerequisites need to be met. Refer to the documentation from Ansible if your operating system is not Debian based. For this lab, our control machine is Debian based. Follow the steps below to get Ansible up and running.

```
 2
 3  sudo apt-get update
 4
 5  sudo apt-get install software-properties-common
 6
 7  sudo apt-add-repository ppa:ansible/ansible
 8
 9  sudo apt-get update
10
11  sudo apt-get install ansible
12
```

- Line 3: This command updates the package managers cache of available software that can be installed. It is important to run this so that line 5 installs the current version of the specified package.
- Line 5: This command downloads and installs scripts that are useful when adding and removing additional repositories to our system.

- Line 7: Using one of the scripts we just downloaded in the previous command, the line adds the Ansible repository to our sources list so that we can easily manage the Ansible package.

- Line 9: Anytime a repository is added or removed from the sources list another update needs to be run against the package managers cache. Now that we have added the Ansible repositories, our package manager still does not know about any of the Ansible packages since it only has a list of what is available based on the last cache update.

- Line 11: Finally, this command uses the package manager to reach out and install Ansible on our control machine.

Exercise 2: Post Installation Tasks

- By default, Ansible manages machines using the SSH protocol. Since SSH has not yet been configured in our environment we need to take a few minutes to create the keys for our user and get them copied over to our target machines. These commands are not Ansible specific, however without them our lab will not work.

- You work environment may use a different method of ensuring keys are on the hosts machines. You are not wrong for doing it your way, this is just ONE of the many ways to distribute the necessary keys. What will ALWAYS be true, however, is that Ansible uses SSH, so ensuring that good key management is in place is essential to adopting this tool.

- To generate a set of keys, use the following command:

```
ssh-keygen
```

- You will be prompted for a passphrase. In our environment **LEAVE IT BLANK** and just press ENTER. In your actual production environment this is not a safe, nor recommended, practice! If you do not leave it blank however, you will need to type that passphrase every time we run anything using Ansible… for each host that task is run against!

- Once you have the key generated we now need to copy it to every single machine in our environment so that we can connect to them using SSH. To use this command, we will need the IP Addresses of our target machines.

- Retrieving our IP Addresses is a little unique in our environment. Since we are using Docker containers we need to use a Docker command to obtain this information. The command is as follows.

```
sudo docker inspect fda | grep IPAddress
```

- Where **fda** is the first three characters of the CONTAINER ID. We will need to do this process for every container in our environment. Refer to the steps on connecting to our lab to see how to list out our containers.

- Once we have the IP Addresses of our other containers we can then copy our keys to them using the following command.

```
ssh-copy-id target_host_ipaddress
```

- You will be prompted to enter the password for the root account. That password is **pass** all lowercase.

Exercise 3: Building an Inventory

- Ansible makes configuration management a breeze for us, however it has no idea what machines to configure unless we give it somewhere to look for those machines. Being agentless, it relies on us telling the control machine what our actual targets are.

- Let's look at an ad-hoc command in ansible. For this we will pick one of our IP Addresses that isn't our Ansible control machine. The first ad-hoc command we will look at uses the ping module. Refer to the screenshot below.

```
1
2  ansible -m ping 172.19.0.3
3
```

- Line 2 is comprised of multiple parts that each play a key role in this ad-hoc command. First **ansible** calls the Ansible program installed on our control machine. The **-m** specifies that we are going to use a module, in this case the module used is **ping**. The **ping** module requires an argument to function properly, in the above example that argument is the IP Address of the node we wish to ping.

- While using Ansible in ad-hoc mode completes our task as desired, the module only ran against one target machine. In our environments we always have an array of targets to complete tasks against. Ansible handles these targets in it's inventory file. The inventory is located, by default, in the path below.

```
1
2  /etc/ansible/hosts
3
```

- Note that the actual file is named **hosts**, do not confuse this with the hosts file that is used to map IP Addresses to host names on a network. Each operating system has a hosts file, and this is largely confusing to new Ansible users. The **hosts f**ile in this directory is Ansibles inventory file.

- Let's take a few minutes to build out the inventory file we are going to use for the time being in this lab. On your control machine navigate to the location of the Ansible hosts file and use vim to open it for editing.

```
1
2  cd /etc/ansible/
3
4  vim hosts
5
```

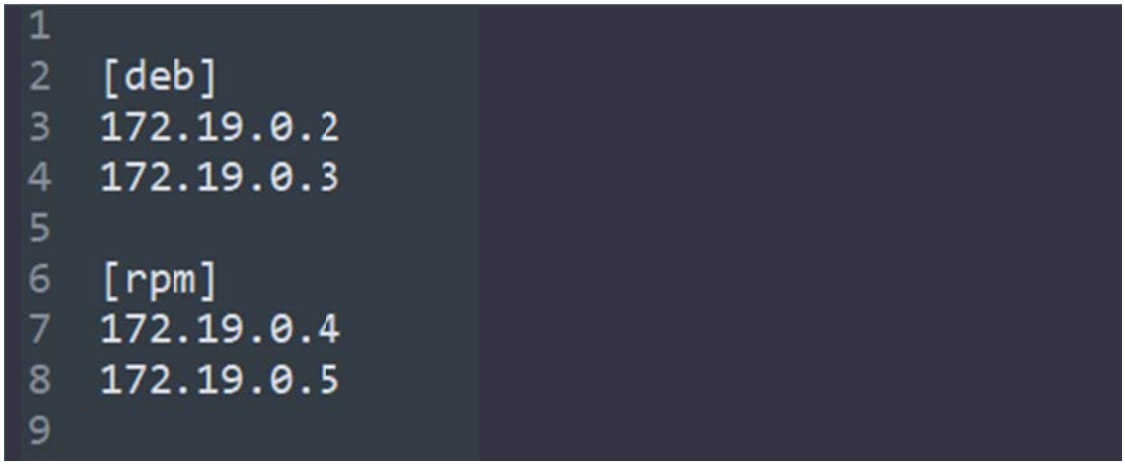
- Line 2 is what sets our working directory to the desired location

- Line 4 uses vim, a terminal-based text editor, to make changes to our hosts file.

- When the **hosts** file is opened for editing you will notice it is prefilled with examples in comment form for the host file. Take a minute to read through what is there. Once you are done you can either remove all the current contents of the **hosts** file, or just navigate to a new line at the bottom of the document.

- Adding the target machines in our environment to the inventory is done in a simple manner. No programming language is necessary, making this an ideal tool for just about anyone in your organization.

- To form a new group inside of the inventory file square brackets, [ ], are used with a desired group name in between them. First, let's make two groups that separate our Ubuntu machines from our Centos7 machines.



- Keep in mind that the IP Addresses you have for your target machines might be different from what you see in the example above. Make sure you place the IP Addresses that your target machines are using into the proper group.

- Line 2 uses the square brackets to declare a group named **deb**. This is name was assigned by us and could have been anything we wanted it to be, **deb** makes sense for our Ubuntu machines, since that's what the distribution is based on.

- Lines 3-4 place one IP Address per line, in this case it is the IP Addresses of our Ubuntu machines.

- Line 6 creates a second group named **rpm** which will contain the IP Addresses of our Centos7 targets. Again, this could be named anything, so use care when naming groups, the more they make sense the easier managing them in the future will be.

- Lines 7-8 add our Centos7 machines to the **rpm** group. Double check to be sure you added the IP Addresses of the Centos7 machines currently running in your environment, as they might be different from what you see in the example above.

- Now that we have two separate groups we can practice using Ansible in ad-hoc mode, and better orchestrating where those modules are being directed. Let's use the **ping** module again, but this time we will specify a group instead of a single IP Address.

```
1
2  ansible -m ping deb
3
```

- You should now see output that shows the result of the module running against the entire group. This is just a small example of how configuration management can be leveraged, there are more to come as this course continues.
- Ansible contains a built-in group. The name of this group is **all**. This group allows us to run a module, or playbook, which will be covered later, against every host, except localhost, in the inventory file. Let's try this same module using **all** instead of one of our newly created groups.

```
1
2  ansible -m ping all
3
```

- There should now be output for every host in your inventory file.

- Now that we have seen the most basic way to use an inventory file, use vim to create two new groups **dbservers** and **webservers**. Once both groups are made add one Ubuntu machine and one Centos7 machine to each group. We will use these two groups as this lab continues, so please be sure to ask for help if you are stuck.

```
1
2  [deb]
3  172.19.0.2
4  172.19.0.3
5
6  [rpm]
7  172.19.0.4
8  172.19.0.5
9
10 [dbservers]
11 172.19.0.2
12 172.19.0.4
13
14 [webservers]
15 172.19.0.3
16 172.19.0.5
17
```

- Take note that now we have a situation where our target machines are members of multiple groups. This is not uncommon for an Ansible inventory file. It is important to understand that if a target is a part of multiple groups, that target will get its variables from all the groups it is a part of.

- The Ansible inventory is also capable of holding behavioral inventory parameters for each host. In this next section we will use the variables to give our target machines actual names, which will allow us to work with them easily. Understand that although this will look like a DNS entry, it is nothing more than a user defined alias for a given target machine.

- We will look at variables in greater detail later in this workshop. For now, let's add the example below to our playbook just so we can get an introduction to the flexibility of the inventory file.

```
ubuntu1 ansible_host=172.19.0.2

[deb]
ubuntu1
172.19.0.3

[rpm]
172.19.0.4
172.19.0.5

[dbservers]
172.19.0.2
172.19.0.4

[webservers]
172.19.0.3
172.19.0.5
```

- Notice how we used the **ansible_host** behavioral inventory parameter to assign the name **ubuntu1** to the IP Address of one of our Ubuntu target machines. Take note that **ubuntu1** is the alias being assigned and is a completely made up name which could have been anything.

- Also take note that we only replaced the IP Address with our alias in the **deb** group. The target machine is still listed in the **dbservers** group with its IP Address. This is NOT best practice but will remain like this for our lab to help demonstrate how to use the Ansible inventory.

- We can also use our newly created alias to run specific Ansible modules. Look at the example below and give it a try.

```
ansible -m ping ubuntu1
```

- For more behavioral inventory parameter consult the documentation at http://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html

Exercise 4: Ad-Hoc Commands

- This exercise will explore using Ansible for one-time ad-hoc commands. Now that we have our inventory setup we can look at some of the power, as well as limitations, ad-hoc commands give us.

- Almost any module that Ansible is capable of utilizing can be run directly from the command line as an ad-hoc command. To list the modules currently available in your environment insert the following into the command line.

```
1
2  ansible-doc -l
3
```

- Now that we know how to see the available modules, let's take a quick look at an individual module. To do that we type the following into the command line.

```
1
2  ansible-doc <module name>
3
4  ansbile-doc user
5
6  ansible-doc apt
7
```

- Before we begin a quick overview of the Ansible command syntax is in order. Examine the screenshot below.

```
1
2  ansible -m <module name> -a <attributes for module> <target machine> <additional args>
3
```

- Ansible does allow for some flexibility with it's syntax. For example, if you wanted to specify the **<target machine>** prior to the module, this syntax would work as well.

- Next, we will work with the **apt** module. This module corresponds to the apt package manager currently installed on the target machine. If there is no apt package manager installed, then this module fails, and the system is not affected.

- A common task that we might need to do across a series of hosts before installing software is to update the repository cache of our target machines. Using Ansible that can be accomplished with the following command.

```
1
2  ansible -m apt -a "update_cache=yes" deb
3
```

- At quick glance we can see that the arguments passed to the **apt** module need to be passed inside of quotations. If we used a second argument, which we will see a bit later, each argument is separated with a single space.

- When this ad-hoc command is executed, it will only affect the target machines in the **deb** group, which in our case is currently two target machines.

- Now that we have updated the package cache on each of our Ubuntu machines, let install our first package. We will use something simple for now, but the complexity will increase as we move forward in this workshop.

```
1
2  ansible -m apt -a "name=vim state=present" deb
3
```

- This time around two arguments get passed to the **apt** module to accomplish this task. First the attribute of **name** is called. This attribute corresponds directly to the package name as it is listed in the repository cache. Secondly, the **state** attribute specifies whether this package should exist on the target machines.

- Let's examine our output before we move further along in this section of our workshop. Below is a screenshot of the output from using the **ping** module.

```
1
2  172.19.0.3 | SUCCESS => {
3      "Changed": false,
4      "ping": "pong"
5  }
6  ubuntu1 | SUCCESS => {
7      "Changed": false,
8      "ping": "pong"
9  }
10
```

- The first portion of the output identifies our target machine. You should notice that one of our machines is listed with its IP Address, while the other machine is using the alias we provided it through a variable in the inventory file that we are currently using.

- SUCCESS is just letting us know that the module completed its task without error. This does NOT mean that a change was made on the target machine. Do not confuse the two.

- "Changed": false, lets us know that nothing on the target machine was changed. In the case of **ping** there is no need to change anything. We will often see a successful module run that results in no change on the target machine.

- Remember, Ansible is idempotent, meaning it only makes changes on the target machines that currently reflect a different configuration than what our desired state is.

- A good way to visualize this is to run the ad-hoc command to install **vim** one more time against our **deb** group. When doing so, the output should look like what you see below.

```
172.19.0.3 | SUCCESS => {
    "cache_update_time": 1522186829,
    "cache_updated": false,
    "changed": false
}
ubuntu1 | SUCCESS => {
    "cache_update_time": 1522186829,
    "cache_updated": false,
    "changed": false
}
```

- Since **vim** was already installed on our target machines the status of **changed** is still equal to **false** since idempotency was already met.

- Let's remove **vim** from our **deb** group targets and look at the difference in output. Use the following command to ensure a package is no longer installed on a target machine.

```
ansible -m apt -a "name=vim state=absent" deb
```

- We can do the same thing to our **rpm** group using the **yum** package manager. As it currently stands our group of targets has **vim** already installed on them. To remove **vim,** use the following command.

```
ansible -m yum -a "name=vim state=absent" rpm
```

- It can become tedious and cumbersome to use Ansible with ad-hoc commands. Later in the workshop we will explore using playbooks to accomplish these tasks in a more automated fashion.

- A list of modules, their uses, attributes, and examples can be found at http://docs.ansible.com/ansible/latest/modules/modules_by_category.html

Exercise 5: Playbooks

- Ad-hoc commands, as we have stated, are a great way to get a quick task accomplished across all your target machines. However, these commands defeat the purpose of achieving a state of idempotency to a degree.

- Consider a situation where a member of your organization uses an ad-hoc command to install software on just a handful of production machines, since it's easy to do so with Ansible, but fails to document the changes. Over time these actions, performed by many members in your organization, will cause so much configuration drift that you end up in the same situations you are in today.

- Playbooks are the answer to mitigating that drift. A good practice is enforcing that all changes made to outside of an Ansible testing environment be placed into a playbook. By doing so, we can ensure that every machine is configured to their desired state and that all changes made are documented, the changes reside in the playbook after all!

- Playbooks are written in YAML and have the extension .yaml or .yml at the end of their filename. If your teams are not familiar with YAML it would be a good idea to take some time and explore it. It is not a hard "language" to use but does come with quite a few "gotcha's". We will not be covering YAML explicitly in this lab guide so make sure you type EVERYTHING EXACTLY AS YOU SEE IT!

- Our first playbook will build off the modules we used earlier **apt** and **yum** and we will use it to install **vim** on each machine. If this seems super basic, it is, but it demonstrates the value of the playbook. We will write more advanced playbooks later in the workshop.

- First create a new file in your current working directory, it should be **/etc/ansible**, called **first_play.yml** and inside that file place the following.

```
1  ---
2
3  - hosts: deb
4
5    tasks:
6    - name: Install vim on Ubuntu targets
7      apt:
8        name: vim
9        state: latest
10
```

- Let's look at this file line by line to see what's going on before moving forward in our workshop.

- The most important line in the YAML example above is line 1. Although it is not entirely mandatory, it is best practice to start ALL YAML files with three dashes residing on the VERY FIRST LINE of the YAML file. What do we mean by not entirely mandatory? Well, sometimes YAML will work without it, sometimes it will not, it really depends on the use case.

- Line 3 specifies the target machines we wish to run the current play of our playbook against. In our case we chose the group of targets named **deb**, but we could have specified a single target as well, or even used the group **all** for this play.

- It is in this section that we could specify behavioral parameters as well, such as becoming a specific user to accomplish the task or specifying whether we need escalated privileges to run the current play.

- Line 5 is where the play really begins! The keyword **tasks** is what denotes the actions that will take place on the targets specified.

- Line 6 is important and very useful. This names our play. It is best practice to name our plays something that is sensible, since it is seen when the play runs, which will be demonstrated shortly. You can name your plays whatever you'd like.

- Line 7 specifies the module to be used. Everything available to use as an ad-hoc command can be placed inside of a playbook, and everything that resides in a playbook can be used as an ad-hoc command. Using a playbook instead of the ad-hoc command allows us to track the changes made on target machines a lot easier since those changes are documented as they are performed.

- Line 8-9 are the parameters for the module specified in line 7. Look back to the ad-hoc section and you will find these same parameters used with the **-a** switch. The difference here is in the **state** attribute. We chose to use latest instead of present. You can read more about differences in these states in the modules section of the Ansible documentation.

- Great! You have now written your first playbook, now it's time to execute that playbook. Use the command below to run your playbook and observe the difference in output.

```
1
2  ansible-playbook -i hosts first_play.yml
3
```

- A quick breakdown of this command is in order since it is significantly different from how we used Ansible in the past.

- The first portion, **ansible-playbook**, tells ansible that a playbook will be used, not an ad-hoc command. Playbooks will not work if you try to use **ansible** as the command, like we did with our ad-hoc commands.

- We now see a new switch, **-i**, which allows us to specify the inventory file we wish to use. If you are in a directory that does not contain an inventory file you will need to specify the path to your desired inventory file after this switch. In our case, we are in the same directory and our inventory file is named **hosts**.

- Lastly, we specify the name of the playbook we wish to run. This is the playbook you created in the previous step.

- Go ahead and execute this command now and examine the output after the playbook completes.

```
1
2  PLAY [deb] *********************************************
3
4  TASK [Gathering Facts] *********************************
5  ok: [172.19.0.3]
6  ok: [ubuntu1]
7
8  TASK    [Install vim on Ubuntu targets] *****************
9  ok: [172.19.0.3]
10 ok: [ubuntu1]
11
12 PLAY RECAP *********************************************
13 172.19.0.3  :ok=2   changed=0   unreachable=0   failed=0
14 ubuntu1     :ok=2   changed=0   unreachable=0   failed=0
15
```

- The output from a playbook is quite nice, and does a great job reporting to us what took place. It tells us the name of the tasks being executed, whether they were successful, indicated with **ok**, and what target machines were checked. Once it completes it shows us a PLAY RECAP which tells us if changes took place, if a host couldn't be reached, or how many plays failed.

- Before we move forward, now is a good time to look at what Ansible means when it says it's "gathering facts". There is quite a list of information that Ansible collects from each target machine, and all these pieces of information can be used as a variable or for building complex logic into our play books.

- Use the following command to see what Ansible gathers on each target machine before running a module.

```
ansible -m setup ubuntu1
```

- Wow that's a lot of information! We can modify this ad-hoc command to filter for specific information. Try it by using the commands below one at a time.

```
ansible -m setup -a "filter=ansible_os_family" ubuntu1

ansible -m setup -a "filter=ansible_os_family" all

ansible -m setup -a "filter=ansible_pkg_mgr" all

ansible -m setup -a "filter=ansible_memory_mb" ubuntu1
```

- In the next phase of our playbook we will make use of the **ansible_os_family** fact that Ansible checks for. We will use this to decide what package manager to use for our play.

- Open the playbook you currently have built with vim and append the following to it. Make note that the desired target group is also going to change in this file. Be sure EVERYTHING is EXACTLY as you see it in the screenshot.

```
---

- hosts: all

  tasks:
  - name: Install vim on Ubuntu targets
    apt:
      name: vim
      state: latest
    when: ansible_os_family == "Debian"

  - name: Install vim on RPM targets
    yum:
      name: vim
      state: latest
    when: ansible_os_family == "RedHat"
```

- A new keyword has appeared in our playbook, **when**, and it does exactly what it sounds like. When this fact is gathered on a system it will execute the task associated with all machines that contain that fact. This is good news for us since our environment is not homogenous, much like your actual environments, but this also means we need to architect our playbooks in a way that these facts don't cause issues.

- In a production world, we would want to be more specific about the groups we are going to run this against, for our labs we chose **all** simply to demonstrate how Ansible will use this logic.

- Save this playbook and execute it with the following command.

```
1
2  ansible-playbook -i hosts first_play.yml
3
```

- This time when the playbook gets executed we are faced with two new pieces of output. First, we see that we have an error, when examined closely this error is a timeout error, more on that in another section. Second, we now see the word **skipped** because of the logic we built in using the **when** keyword. If Ansible cannot configure a machine for any reason it will skip it and tell you the command, you need to run to try again. In our case, we can run that command an infinite number of times and nothing will happen because the **ansible_os_family** does not match, so therefore the task will not execute.

- This concludes working with basic playbooks. We will expand on this in future exercises.

Exercise 6: Advanced Inventory and Variables

- In this section we will begin to explore some advanced features of the inventory file, organization of multiple inventory files, expand on our knowledge of variables, and fix the timeout error we encountered when working with playbooks.

- In the first portion of this exercise we will fix the timeout error we encountered while working with our playbook.

- This error occurred because our **ubuntu1** machine was also listed as **172.19.0.2** in another group. While this is perfectly fine when using an ad-hoc command, the playbook gathers facts on all the specified targets at once, before executing a single task.

- Because of this, Ansible was trying to gather facts about the same target machine, twice, at the same time, causing that target to not report back in the timeframe allotted by Ansible, therefore throwing a timeout error.

- One fix for this is to edit our inventory file. While we are fixing this issue lets add a few more behavioral inventory parameters. Edit your inventory to match the screenshot below.

```
1
2  ubuntu1 ansible_host=172.19.0.2
3  ubuntu2 ansible_host=172.19.0.3
4  centos1 ansible_host=172.19.0.4
5  centos2 ansible_host=172.19.0.5
6
7  [deb]
8  ubuntu1
9  ubuntu2
10
11 [rpm]
12 centos1
13 centos2
14
15 [dbservers]
16 ubuntu1
17 centos1
18
19 [webservers]
20 ubuntu2
21 centos2
22
```

- Once you have done that go ahead and run your playbook again, this time there should be no errors because the facts are only being gathered one time since our host is in our inventory file in a more consistent manner.

```
1
2  ansible-playbook -i hosts first_play.yml
3
```

- We should now see the following output, which has no timeout error and is now more human readable since we see target names and not just IP Addresses.

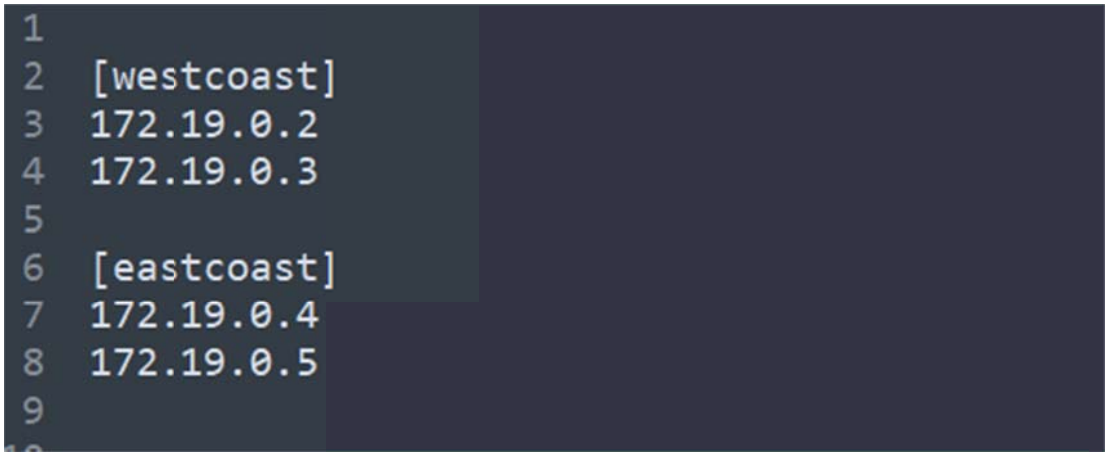
```
PLAY [all] *********************************************

TASK [Gathering Facts] *********************************
ok: [ubuntu2]
ok: [ubuntu1]
ok: [centos2]
ok: [centos1]

TASK [Install vim on Ubuntu targets] *******************
skipping: [centos1]
skipping: [centos2]
ok: [ubuntu2]
ok: [ubuntu1]

TASK [Install vim on Ubuntu targets] *******************
skipping: [ubuntu1]
skipping: [ubuntu2]
ok: [centos2]
ok: [centos1]

PLAY RECAP *********************************************
centos1     :ok=2   changed=0   unreachable=0   failed=0
centos2     :ok=2   changed=0   unreachable=0   failed=0
ubuntu1     :ok=2   changed=0   unreachable=0   failed=0
ubuntu2     :ok=2   changed=0   unreachable=0   failed=0
```

- Next, we will look at using multiple inventory files within ansible. Use your text editor to create a file named **organization**. Inside of that file create a new group named **westcoast** and inside that group place your two Ubuntu target machines.

```
[westcoast]
172.19.0.2
172.19.0.3
```

- Now add another group named **eastcoast** and place your Centos7 target machines in that group.

- Your new inventory file, named **organization** should look like the following screen capture.



- Notice how in this scenario we are not including any behavioral inventory parameters. Be sure to leave these out in your new inventory file as well for consistency in the workshop.

- On our Ansible control machine there are now two inventory files. The first, named **hosts**, is created by default but is blank when we install Ansible. The second, **organization**, is one that we just created. For our lab environment we reused the same target machines in each inventory file, this is NOT a requirement, your new inventory file could contain target machines that are unique to it. You can have as many inventory files as makes sense for your environment.

- Using Ansible with multiple inventory files is easy, we just must tell Ansible which inventory file we would like to use. Using the **-i** switch, which we saw when calling **ansible-playbook**, will allow us to specify which inventory file we want the actions to be performed against. Run the following command from your terminal.

```
ansible -m ping -i organization westcoast
```

- Although this is an ad-hoc command, Ansible still allows us to specify a non-default inventory file. We would use the same syntax when running a playbook as well.

- Separating inventory files is one way to help organize your target machines in a more sensible manner, but there is more we can do. Next, we will learn more about groups inside of an inventory file.

- Before we get started let's remove the inventory file named **organization** to simplify things for this lab. To do so use the following command.

```
1
2  rm organization
3
```

- Now that our second inventory file is no longer on our system, we will make some edits to the default inventory file, named **hosts**. Use the following screen capture to guide you in making these changes.

```
1
2  [group1]
3  172.19.0.2
4  172.19.0.3
5
6  [group2]
7  172.19.0.4
8  172.19.0.4
9
10 [nodes:children]
11 group1
12 group2
13
```

- We will take a quick look at what we just did before moving forward in this workshop.

- Line 2 and 6 are things we have seen before at this point. If you recall from our first exercise involving inventory files, these are simply group names that we assigned based on what we thought a logical name for our group would be.

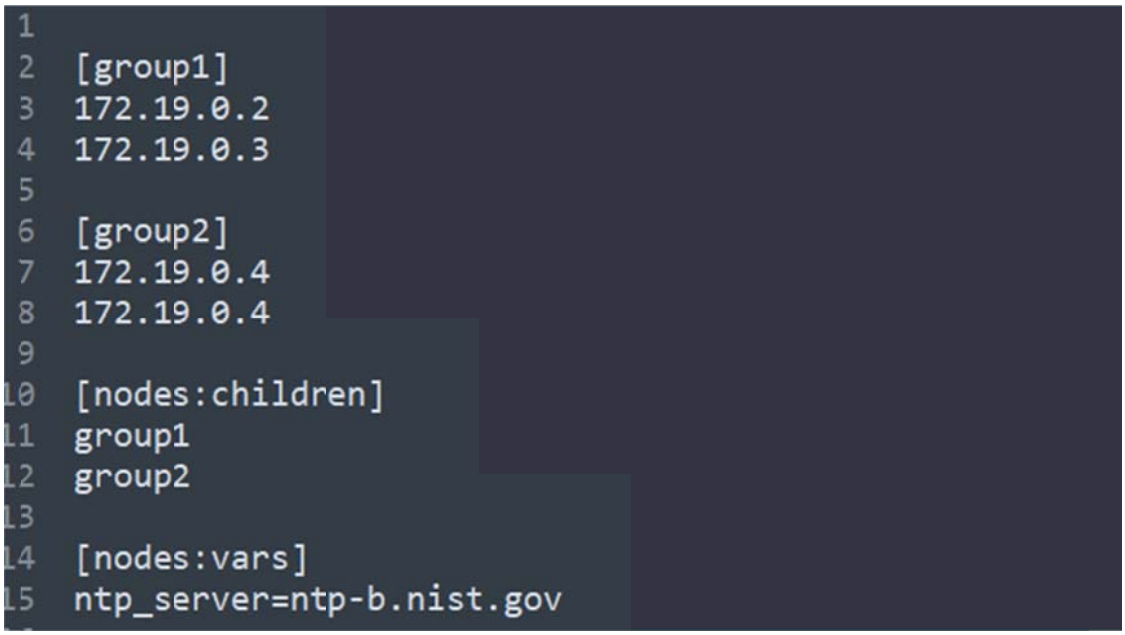
- Line 10 shows us something new, the keyword **children**. Ansible uses this keyword inside an inventory file to allow us to create a group of groups. If you chose to do so, you could even group multiple groups of groups to help organize your environment better for your organization.

- So how does this all work out? I'm glad you asked! Use the command below to see how running modules against a group of groups works.

```
1
2  ansible -m ping nodes
3
```

- This is nothing new, however, we can see that we get responses from all four of our target machines even though they reside in different groups, and we reference those groups by providing the parent group name, in this case **nodes**, which could be any name you wanted to give it such as geographical location of your target machines, or role they play in the environment.

- Groups of groups are just one way we can leverage the inventory file to better fit a specific use case scenario. Another thing we can do is assign variables to groups inside of our inventory file. In this example, we will set a NTP server variable for all our target machines. Add the following to your inventory file.



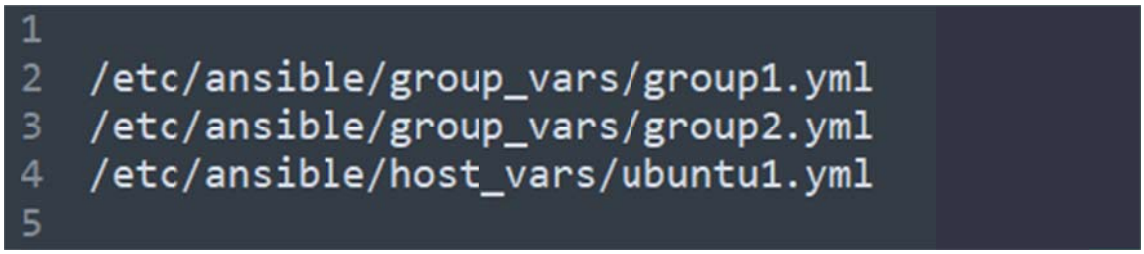
- On line 15 we used the **vars** keyword to assign variables to our target machines in the group of nodes.

- If you recall in the first inventory exercise we assigned behavioral inventory parameters to a host, but just like group variables, we could do the same for a specific host.

- This is just one way to assign variables but may not be the best way for your situation. Next, we will look at breaking our variables out into different directories and placing them outside of our inventory file.

- Unfortunately, there is not a huge use case for this in our lab environment, so for the purposes of this workshop we will just be giving a brief overview of how to separate group variables into directories.

- In the case of our current host file, see the screen capture below for what it would look like if we wanted to separate out variables.



- On lines 2 and 3 we have created a directory in our current working directory named **group_vars**, and inside that directory we have created a YAML file that corresponds to the group name in our inventory file. It is in these YAML files that we place our variables. In the case of group variables, you will need one YAML file per group.

- On line 4, we did the same thing, only this time the directory name is **host_vars** and the file containing the variables is in the YAML file with the name of a specific host, which was set with a behavioral inventory parameter. In the case of host variables, you will need one YAML file per host.

- A few key points need to be mentioned at this current moment. If you plan to separate your variables in directories like the screen capture above, they MUST be written in YAML.

- Also, if a host variable is set, but that host is also in a group, ALL the variables in the applicable YAML files will be applied to the host.

- This concludes this section of labs. Be sure to let your instructor know if there is anything you do not understand at this point before moving forward in the workshop.

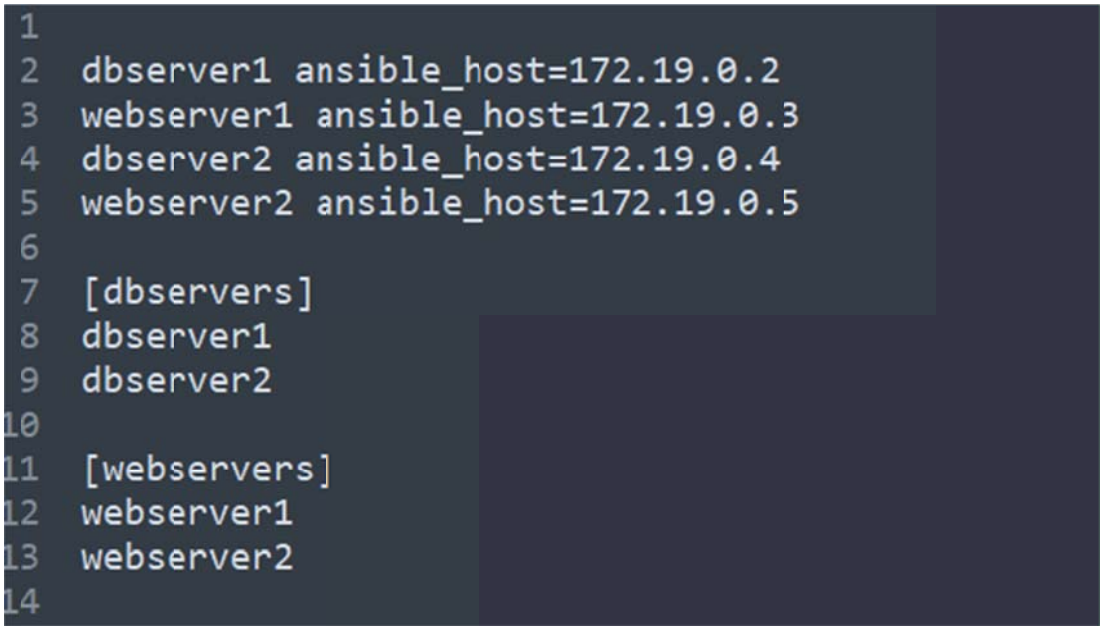
- For more information on variables and advanced inventory features, consult this section of the documentation
http://docs.ansible.com/ansible/latest/user_guide/intro_inventory.html#groups-of-groups-and-group-variables

Exercise 7: Advanced Playbooks

- In an earlier section we took a brief look at playbooks and used our first playbook to simply install a text editor across all our machines. This was a very basic task but helped us see the power we can leverage with Ansible.

- In this section we are going to dig deeper into writing more complex playbooks which will allow further understanding of the capabilities Ansible has in our environments.

- We will start of by making some changes to our inventory file, named **hosts**, in our current working directory **/etc/ansible**. You inventory file should match the screen capture below.



- Take care to ensure that each group, **dbservers** and **webservers** contain one Ubuntu machine as well as one Centos7 machine.

- Now that we have that complete we can start writing our playbook.

- What good is a webserver group unless we install a webserver onto those machines? This is going to be our first task in the playbook. Use your text editor to create a file named **second_plays.yml** and inside that file place the following information.

```
1  ---
2
3  - hosts: webservers
4    vars:
5      http_port: 80
6    remote_user: root
7
8    tasks:
9    - name: Install Apache webserver deb
10     apt:
11       name: apache2
12       state: latest
13     when: ansible_os_family == "Debian"
14
15   - name: Install Apache webserver rpm
16     yum:
17       name: httpd
18       state: latest
19     when: ansible_os_family == "RedHat"
20
21   - name: Start Apache service (enable upon boot) deb
22     service:
23         name: apache2
24         state: started
25         enabled: yes
26     when: ansible_os_family == "Debian"
27
28   - name: Start Apache service (enable upon boot) rpm
29     service:
30         name: httpd
31         state: started
32         enabled: yes
33     when: ansible_os_family == "RedHat"
34
```

- As usual, we are seeing a few new things at this point, so let's pause for a second to go over these new items before moving on.

- On line 4 we see the **vars** keyword. Like an inventory file, our playbooks can also contain variables. We add these right after the specified group of hosts that the play pertains to. This same variable could have been placed in the inventory file, so it's largely up to you and your organization to determine where you would like to place them. Since it's a webserver, we made sure that port 80 was accessible with this variable.

- Line 6 allows us to run a play as a specific user. This could have also been assigned to specific tasks within the play, we could have also specified a regular user with administrator privileges and use privilege escalation to accomplish a full play or individual task.

- The rest of this playbook should look familiar to you. The exception is now we are using Ansible to manipulate services running on the target machine. In our case we used it to start and enable our webserver. You probably gathered that from the task names however, which is another reason to name your tasks, although it is not required.

- Use the following command to run your play.

```
ansible-playbook -i hosts second_plays.yml
```

- If everything ran correctly, you output should look like the screen capture below.

```
PLAY [webservers] ***************************************************

TASK [Gathering Facts] **********************************************
ok: [webserver1]
ok: [webserver2]

TASK [Install Apache webserver deb] *********************************
skipping: [webserver2]
ok: [webserver1]

TASK [Install Apache webserver rpm] *********************************
skipping: [webserver1]
ok: [webserver2]

TASK [Start Apache service (enable upon boot) deb] ******************
skipping: [webserver2]
ok: [webserver1]

TASK [Start Apache service (enable upon boot) rpm] ******************
skipping: [webserver1]
ok: [webserver2]


PLAY RECAP **********************************************************
webserver1      :ok=3   changed=2   unreachable=0   failed=0
webserver2      :ok=3   changed=2   unreachable=0   failed=0

```

- Notice that in the PLAY RECAP there are three things that passed as **ok** even though our playbook only contained two tasks. The reason for this is the fact gathering that Ansible does. Ansible see's GATHERING FACTS as a task on its own.

- We also see that two changes have taken place. These two changes are the installation of Apache and the manipulation of the services.

- So how do we know that line 6 of our playbook was executed? We can ask Ansible to give us more verbosity on what it is doing. Try running your play with the following switch enabled and see if you can spot where line 6 is used in the output. Also take note that now you should see zero changes since are targets are already in their desired state.

```
1
2  ansible-playbook -i hosts second_plays.yml -vvv
3
```

- A playbook is a collection of plays. Right now, we only have one play, so let's take this opportunity to setup another play to configure our database servers in the same playbook. Use vim to edit **second_plays.yml** and add the following.

```
28    - name: Start Apache service (enable upon boot) rpm
29      service:
30          name: httpd
31          state: started
32          enabled: yes
33      when: ansible_os_family == "RedHat"
34
35  - hosts: dbserver2
36    gather_facts: no
37
38    tasks:
39      - name: Install MongoDB
40        yum:
41          name: mongodb-server
42          state: present
43        notify:
44          - start mongodb
45
46    handlers:
47      - name: start mongodb
48        service:
49          name: mongod
50          state: started
51
```

- This time around instead of specifying an entire group that will utilize our play, we decided to pick just one of our target machines. Since we know exactly what this machine has for an operating system we chose to turn off the fact gathering that Ansible typically does on line 36. Turning this off saves us network bandwidth as well as speeds up the execution of our playbook, however we lose the ability to use any facts for our playbook logic.

- Another new keyword we see is **notify**. When this keyword is used a **handler,** gets called when the current task finishes its execution. If multiple tasks call the same **handler** Ansible waits until the last one is called and only runs the **handler** one time to avoid repeating tasks unnecessarily.

- What we chose to name our notifier and our handler was completely up to us, since this handler was going to start the mongodb service we chose to name it **start mongodb**.

- Line 46 specifies what task should run when we executed the handler. A thing to know about handlers is that they only run if there is a change to the task that notifies the handler. For example, if we copied a new configuration file to our target machine and had a handler designed to restart a service, that copied file would trigger the handler.

- On the contrary, if that configuration file already existed on our target machine the handler would NOT be notified since no actual change in the target machines state would take place.

- We are starting to make a lot of changes to our playbook at this point, changes that could potentially cause problems in production. Luckily Ansible has a way for us to run a simulation of our playbook. If we add a switch to the **ansible-playbook** command Ansible will shows us output as if it ran the playbook so that we can see what WOULD have happened if we ran it without the switch.

- Before we look at that command, it would be helpful to know that NOT ALL tasks run as simulations with this switch. For example, the **command** module will still execute on the desired targets even when **ansible-playbook** is running in simulation mode. So, use with extreme caution if these types of tasks exist in your play book.

- To run a simulation, use the following switch along with **ansible-playbook**.

```
1
2  ansible-playbook -i hosts second_plays.yml --check
3
```

- Go ahead and run your new playbook with this switch enabled and examine the output.

- When looking at the output below, notice that the **handler** would have executed during this run of the playbook. That's because mongodb does not currently exist on our dbserver-to-be. Once we run this playbook without the simulation enabled we will only see this output if a change occurs.

```
PLAY [webservers] ***************************************************

TASK [Gathering Facts] **********************************************
ok: [webserver1]
ok: [webserver2]

TASK [Install Apache webserver deb] *********************************
skipping: [webserver2]
ok: [webserver1]

TASK [Install Apache webserver rpm] *********************************
skipping: [webserver1]
ok: [webserver2]

TASK [Start Apache service (enable upon boot) deb] ******************
skipping: [webserver2]
ok: [webserver1]

TASK [Start Apache service (enable upon boot) rpm] ******************
skipping: [webserver1]
ok: [webserver2]

PLAY [dbserver2] ****************************************************

TASK [Install MongoDB] **********************************************
changed: [dbserver2]

RUNNING HANDLER [start mongodb] *************************************
changed: [dbserver2]

PLAY RECAP **********************************************************
dbserver2       :ok=2   changed=2   unreachable=0   failed=0
webserver1      :ok=3   changed=0   unreachable=0   failed=0
webserver2      :ok=3   changed=0   unreachable=0   failed=0

```

- Notice in this simulation that our **dbserver2** play only show two tasks being ok. This is because we skipped the gathering facts phase of using an Ansible playbook.

- Since this was a simulation none of the output resembles a change on the target machines. Go ahead and run **second_plays.yml** without adding the switch for a simulation. To do so use the following command.

```
ansible-playbook -i hosts second_plays.yml
```

- Take note that the output is the exact same as we encountered with our dry-run, only this time the changes took place on our target machines.

- We still have one database server that needs to be configured, and this is a great opportunity to explore even more option within our playbook. Edit **second_plays.yml** and append the following to it.

```
46   handlers:
47     - name: start mongodb
48       service:
49         name: mongod
50         state: started
51
52 - hosts: dbserver1
53   gather_facts: False
54
55   tasks:
56     - name: Install MySQL
57       apt:
58         update_cache: yes
59         name: mysql-server
60         state: latest
61
62     - name: Get MySQL Status
63       shell: service mysql status >> ~/mysqlstats.txt
64       args:
65         warn: no
66       ignore_errors: yes
67       tags:
68         - mysqlstatus
69
70     - name: Deliver MySQL Status
71       fetch:
72         src: ~/mysqlstats.txt
73         dest: /etcansible/fetchedfiles
74         flat: yes
75       tags:
76         - mysqlstatus
77
```

- As we add this play to our playbook we get to look at some new items. On line 58 we see something that we used in the past with an ad-hoc command. The **update_cache** attribute of the **apt** module. This demonstrates that each module attribute can be built into a playbook as well.

- Line 63 shows us a new module, **shell**, which is used to execute shell commands on the target machine. Ansible allows the use of either **shell** or **command** to accomplish these tasks, however there are slight differences and it is worth reading about to understand when to use each module.

- Ansible has some arguments that we can give to modules, which is what we are doing on line 64. This argument tells Ansible to not warn us if there are suggestions for modules to use. In this use case Ansible will tell us that we might want to use the **service** module to complete our task and only use **shell** if for some reason that module was insufficient. Turns out, for what we are trying to do, poll a services status, the service module is not sufficient. With **warn** set to no Ansible will not display this warning.

- Our first look at error handling happens on line 66. Without this argument, this play throws an error and the entire playbook stops, skipping the next tasks altogether. In our use case, the module performs as it should, however Ansible does not like the exit code that this command gives, which is ultimately what stops the playbook. When we set **ignore-errors** to **yes** Ansible will continue to run the playbook, and it will still show us the error.

- We also use a module called **fetch**, to retrieve a file from a remote host and deliver it to our target machine. We will look at what is in that file in a second.

- The **fetch** module has an attribute called **flat** which you can also read up on in the Ansible documentation.

- Lastly, on lines 67 and 75, we see something called **tags**. Tags allow us to run just a specific portion of a playbook. We will look at using tags in a few moments. For now, let's go ahead and run our playbook. Make note that when we run our playbook this time, the handlers that we setup when installing MongoDB will NOT run this time, since no change to that target machine is taking place.

```
1
2  ansible-playbook -i hosts second_plays.yml
3
```

- Running the above command, as you know by now, runs our playbook. If you have any questions at this point please be sure to ask an instructor!

- Looking at our output we can clearly see the error that is thrown from Ansible expecting a different exit code from our shell command, we can also see that it ignored this and continued with the playbook.

- Let's verify that our **shell** task did run. Use the following command to verify that we received the file from the target machine and that it contains the status of the MySQL service.

```
1
2  cat fetchedfiles
3
```

- If successful, you should see the following on your terminal after running the above command.

```
1
2  * MySQL is stopped.
3
```

- Clearly showing us that although Ansible did not appreciate the exit code that was returned to it, the task at hand did do its intended job.

- Before we wrap this exercise up, lets examine how Ansible uses tags. The first step for this is to remove the file that already exists named **fetchedfiles**. Because of how the **fetch** module works, if this file stays in our current location it will fail. To remove it, use the following command.

```
1
2  rm fetchedfiles
3
```

- With a clean directory for **fetch** to run its job, we can now execute the two tasks in our playbook that have the tag **mysqlstatus**. The following command will run just those tasks and nothing more from our playbook.

```
1
2  ansible-playbook -i hosts second_plays.yml --tags "mysqlstatus"
3
```

- Don't be alarmed that Ansible still gathers facts for the webservers group, this is default behavior. Once all of the facts are gathered you will see that every other task is skipped and only the two containing the tag mysqlstatus execute.

- You can verify the successful operation by reading the fetchedfiles text file. Do so using the following.

```
1
2  cat fetchedfiles
3
```

- This time you should see two entries in this file, which happens because of how the **shell** module command is written, giving us a good demonstration of the functionality of tags.

- Tags allow us to either run a specific task or tasks in a playbook, but also allow us to skip specific tasks by using a different switch.

- This concludes our section on advanced playbook usage. This section introduced us to a lot more functionality that an Ansible playbook is capable of, however this is just the tip of the iceberg. As usual, it is recommended to consult the Ansible documentation for further examples and playbook usage.

Challenge yourself with the following:

- Some of these challenges may not be able to be completed due to the nature of our lab environment. Our Docker containers may not have the required setup, but feel free to try anyway.

1. A new developer has been hired by your organization. Write a playbook that will create a user account for this individual, as well as setup their environment with all required software, language libraries, and environmental variables.
2. A new administrator has been hired by your organization. Write a playbook that will create their user account on your local machine, then generate and copy that users SSH keys to all the hosts. Verify that you can utilize SSH to connect to each target machine.
3. Create an Ansible playbook to configure NTP across our environment. Use tags in this playbook to sync the service as drift occurs.
4. Call a playbook you've created using a playbook.
5. Setup and configure a user for our MongoDB target machine.
6. Write a playbook to configure the Apache2 webserver in our environment.

- This concludes our lab handout. Please ensure you understand the topics covered and ask your instructor if you have questions. Feedback is greatly appreciated.