Lab 15. Defining Your Inventory

In this lab, we will cover the following topics:

- Creating an inventory file and adding hosts
- Special host management using patterns

Lab Environment

All lab file are present at below path. Run following command in the terminal first before running commands in the lab:

```
cd ~/Desktop/gitlab-ci-ansible-course/Lab_15
```

Creating an inventory file and adding hosts

Most Ansible installations will look for a default inventory file in [/etc/ansible/hosts] (though this path is configurable in the Ansible configuration file). Most Ansible commands use the [-i] flag to specify the location of the inventory file if not using the default. Hypothetically, this might look like the following example:

```
ansible -i /root/Desktop/gitlab-ci-ansible-course/Lab_15/my_inventory all -m ping
```

In this lab, we will provide some examples of both INI and YAML formatted inventory files. Let's start by creating a static inventory file. This inventory file will be separate from the default inventory.

Create an inventory file in [/etc/ansible/my_inventory] using the following INI-formatted code:

```
target1.example.com ansible_host=127.0.0.1 ansible_port=22

target2.example.com ansible_port=22 ansible_user=root

target3.example.com ansible_host=127.0.0.1 ansible_port=22
```

In this way, even with no further constructs, you can begin to see the power of static INI-formatted inventories.

Now, if you wanted to create exactly the same inventory as the preceding, but this time, format it as YAML, you would specify it as follows. Create an inventory file in [/etc/ansible/my_inventory.yaml] using the following yaml:

```
---
ungrouped:
  hosts:
    target1.example.com:
      ansible_host: 127.0.0.1
      ansible_port: 22
    target2.example.com:
      ansible_port: 22
      ansible_user: root
    target3.example.com:
      ansible_host: 127.0.0.1
      ansible_port: 22
```

Note: You can copy file from Lab_15 folder as well.

```
cd ~/Desktop/gitlab-ci-ansible-course/Lab_15 && cp my_inventory.yaml /etc/ansible/
```

Now if you were to run the preceding inventory within Ansible, using a simple [shell] command, the result would appear as follows:

```
ansible -i /etc/ansible/my_inventory.yaml all -m shell -a 'echo hello-yaml' -f 5


target1.example.com | CHANGED | rc=0 >>
hello-yaml
target2.example.com | CHANGED | rc=0 >>
hello-yaml
target3.example.com | CHANGED | rc=0 >>
hello-yaml
```

That covers the basics of creating a simple static inventory file. Let's now expand upon this by adding host groups into the inventory in the next part of this lab.

Using host groups

Let's assume you have a simple three-tier web architecture, with multiple hosts in each tier for high availability and/or load balancing. The three tiers in this architecture might be the following:

- Frontend servers
- Application servers
- Database servers

Let's, first of all, create the inventory for the three-tier frontend using the INI format. We will call this file [hostsgroups-ini], and the contents of this file should look something like this:

```
loadbalancer.example.com

[frontends]
frt01.example.com
frt02.example.com

[apps]
app01.example.com
app02.example.com

[databases]
dbms01.example.com
dbms02.example.com
```

In the preceding inventory, we have created three groups called [frontends], [apps], and [databases]. Note that, in INI-formatted inventories, group names go inside square braces. Under each group name goes the server names that belong in each group, so the preceding example shows two servers in each group. Notice the outlier at the top, [loadbalancer.example.com]---this host isn't in any group. All ungrouped hosts must go at the very top of an INI-formatted file.

We could, of course, handle this case using facts gathered from each host as these will contain the operating system details. We could also create a new version of the inventory, as follows:

```
loadbalancer.example.com

[frontends]
```

```
frt01.example.com
frt02.example.com

[apps]
app01.example.com
app02.example.com

[databases]
dbms01.example.com
dbms02.example.com

[centos:children]
apps
databases

[ubuntu:children]
frontends
```

With the use of the [children] keyword in the group definition (inside the square braces), we can create groups of groups; hence, we can perform clever groupings to help our playbook design without having to specify each host more than once.

This structure in INI format is quite readable but takes some getting used to when it is converted into YAML format. The code listed next shows the YAML version of the preceding inventory---the two are identical as far as Ansible is concerned, but it is left to you to decide which format you prefer working with:

```
all:
  hosts:
    loadbalancer.example.com:
  children:
    centos:
      children:
        apps:
          hosts:
            app01.example.com:
            app02.example.com:
        databases:
          hosts:
            dbms01.example.com:
            dbms02.example.com:
    ubuntu:
      children:
        frontends:
          hosts:
            frt01.example.com:
            frt02.example.com:
```

Note: You can copy inventory file from Lab_15 folder as well.

```
cd ~/Desktop/gitlab-ci-ansible-course/Lab_15 && cp hostgroups-yml
/etc/ansible/my_inventory.yaml
```

When you want to work with any of the groups from the preceding inventory, you would simply reference it either in your playbook or on the command line. For example, in the last section we ran, we can use the following:

```
ansible -i /etc/ansible/my_inventory.yaml all -m shell -a 'echo hello-yaml' -f 5
```

Note the [all] keyword in the middle of that line. That is the special [all] group that is implicit in all inventories and is explicitly mentioned in your previous YAML example. If we wanted to run the same command, but this time on just the [centos] group hosts from the previous YAML inventory, we would run this variation of the command:

```
ansible -i hostgroups-yml centos -m shell -a 'echo hello-yaml' -f 5

app01.example.com | CHANGED | rc=0 >>
hello-yaml
app02.example.com | CHANGED | rc=0 >>
hello-yaml
dbms01.example.com | CHANGED | rc=0 >>
hello-yaml
dbms02.example.com | CHANGED | rc=0 >>
hello-yaml
```

As you can see, this is a powerful way of managing your inventory and making it easy to run commands on just the hosts you want to. The possibility of creating multiple groups makes life simple and easy, especially when you want to run different tasks on different groups of servers.

As an aside to developing your inventories, it is worth noting that there is a quick shorthand notation that you can use to create multiple hosts. Let's assume you have 100 app servers, all named sequentially, as follows:

```
[apps]
app01.example.com
app02.example.com
...
app99.example.com
app100.example.com
```

This is entirely possible, but would be tedious and error-prone to create by hand and would produce some very hard to read and interpret inventories. Luckily, Ansible provides a quick shorthand notation to achieve this, and the following inventory snippet actually produces an inventory with the same 100 app servers that we could create manually:

```
[apps]
app[01:100].prod.com
```

It is also possible to use alphabetic ranges as well as numeric ones---extending our example to add some cache servers, you might have the following:

```
[caches]
cache-[a:e].prod.com
```

This is the same as manually creating the following:

```
[caches]
cache-a.prod.com
cache-b.prod.com
cache-c.prod.com
```

```
cache-d.prod.com
cache-e.prod.com
```

Now that we've completed our exploration of the various static inventory formats and how to create groups (and indeed, child groups), let's expand in the next section on our previously brief look at host variables.

Adding host and group variables to your inventory

Let's build on our previous three-tier example and suppose that we need to set two variables for each of our two frontend servers. These are not special Ansible variables, but instead are variables entirely of our own choosing, which we will use later on in the playbooks that run against this server. Suppose that these variables are as follows:

- [https_port], which defines the port that the frontend proxy should listen on
- [lb_vip], which defines the FQDN of the load-balancer in front of the frontend servers

Let's see how this is done:

1. We could simply add these to each of the hosts in the [frontends] part of our inventory file, just as we did before with the Ansible connection variables. In this case, a portion of our INI-formatted inventory might look like this:

```
[frontends]
frt01.example.com https_port=8443 lb_vip=lb.example.com
frt02.example.com https_port=8443 lb_vip=lb.example.com
```

If we run an ad hoc command against this inventory, we can see the contents of both of these variables:

```
ansible -i hostvars1-hostgroups-ini frontends -m debug -a "msg=\"Connecting to {{
lb_vip }}, listening on {{ https_port }}\""

frt01.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8443"
}
frt02.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8443"
}
```

This has worked just as we desired, but the approach is inefficient as you have to add the same variables to every single host.

2. Luckily, you can assign variables to a host group as well as to hosts individually. If we edited the preceding inventory to achieve this, the [frontends] section would now look like this:

```
[frontends]
frt01.example.com
frt02.example.com

[frontends:vars]
https_port=8443
lb_vip=lb.example.com
```

Notice how much more readable that is? Yet, if we run the same command as before against our newly organized inventory, we see that the result is the same:

```
ansible -i groupvars1-hostgroups-ini frontends -m debug -a "msg=\"Connecting to {{
lb_vip }}, listening on {{ https_port }}\""

frt01.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8443"
}
frt02.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8443"
}
```

3. There will be times when you want to work with host variables for individual hosts, and times when group variables are more relevant. It is up to you to determine which is better for your scenario; however, remember that host variables can be used in combination. It is also worth noting that host variables override group variables, so if we need to change the connection port to [8444] on the [frt01.example.com] one, we could do this as follows:

```
[frontends]
frt01.example.com https_port=8444
frt02.example.com

[frontends:vars]
https_port=8443
lb_vip=lb.example.com
```

Now if we run our ad hoc command again with the new inventory, we can see that we have overridden the variable on one host:

```
ansible -i hostvars2-hostgroups-ini frontends -m debug -a "msg=\"Connecting to {{
lb_vip }}, listening on {{ https_port }}\""


frt01.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8444"
}
frt02.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8443"
}
```

Of course, doing this for one host alone when there are only two might seem a little pointless, but when you have an inventory with hundreds of hosts in it, this method of overriding one host will suddenly become very valuable.

4. Just for completeness, if we were to add the host variables we defined previously to our YAML version of the inventory, the [frontends] section would appear as follows (the rest of the inventory has been removed to save space):

```
        frontends:
          hosts:
            frt01.example.com:
              https_port: 8444
            frt02.example.com:
          vars:
```

```
            https_port: 8443
            lb_vip: lb.example.com
```

Running the same ad hoc command as before, you can see that the result is the same as for our INI-formatted inventory:

```
ansible -i hostvars2-hostgroups-yml frontends -m debug -a "msg=\"Connecting to {{
lb_vip }}, listening on {{ https_port }}\""

frt01.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8444"
}
frt02.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8443"
}
```

5. So far, we have covered several ways of providing host variables and group variables to your inventory; however, there is another way that deserves special mention and will become valuable to you as your inventory becomes larger and more complex.

Let's start by creating a new directory structure for this purpose:

```
mkdir vartree
cd vartree
```

6. Now, under this directory, we'll create two more directories for the variables:

```
mkdir host_vars group_vars
```

7. Now, under the [host_vars] directory, we'll create a file with the name of our host that needs the proxy setting, with [.yml] appended to it (that is, [frt01.example.com.yml]). This file should contain the following:

```
---
https_port: 8444
```

8. Similarly, under the [group_vars] directory, create a YAML file named after the group to which we want to assign variables (that is, [frontends.yml]) with the following contents:

```
---
https_port: 8443
lb_vip: lb.example.com
```

9. Finally, we will create our inventory file as before, except that it contains no variables:

```
loadbalancer.example.com

[frontends]
frt01.example.com
frt02.example.com

[apps]
app01.example.com
app02.example.com
```

```
[databases]
dbms01.example.com
dbms02.example.com
```

Just for clarity, your final directory structure should look like this:

```
 tree
.
├── group_vars
│   └── frontends.yml
├── host_vars
│   └── frt01.example.com.yml
└── inventory

2 directories, 3 files
```

10. Now, let's try running our familiar ad hoc command and see what happens:

```
cd /root/Desktop/gitlab-ci-ansible-course/Lab_15/vartree
ansible -i inventory frontends -m debug -a "msg=\"Connecting to {{ lb_vip }},
listening on {{ https_port }}\""

frt02.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8443"
}
frt01.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8444"
}
```

As you can see, this works exactly as before, and without further instruction, Ansible has traversed the directory structure and ingested all of the variable files.

11. If you have many hundreds of variables (or need an even finer-grained approach), you can replace the YAML files with directories named after the hosts and groups. Let's recreate the directory structure but now with directories instead:

```
tree
.
├── group_vars
│   └── frontends
│       ├── https_port.yml
│       └── lb_vip.yml
├── host_vars
│   └── frt01.example.com
│       └── main.yml
└── inventory
```

Notice how we now have directories named after the [frontends] group and the [frt01.example.com] host? Inside the [frontends] directory, we have split the variables into two files, and this can be incredibly useful for logically organizing variables in groups, especially as your playbooks get bigger and more complex.

Even with more finely divided directory structure, the result of running the ad hoc command is still the same:

```
ansible -i inventory frontends -m debug -a "msg=\"Connecting to {{ lb_vip }},
listening on {{ https_port }}\""

frt01.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8444"
}
frt02.example.com | SUCCESS => {
    "msg": "Connecting to lb.example.com, listening on 8443"
}
```

12. One final thing of note before we conclude this lab is if you define the same variable at both a group level and a child group level, the variable at the child group level takes precedence. This is not as obvious to figure out as it first sounds. Consider our earlier inventory where we used child groups to differentiate between CentOS and Ubuntu hosts---if we add a variable with the same name to both the [ubuntu] child group and the [frontends] group (which is a **child** of the [ubuntu] group) as follows, what will the outcome be? The inventory would look like this:

```
loadbalancer.example.com

[frontends]
frt01.example.com
frt02.example.com

[frontends:vars]
testvar=childgroup

[apps]
app01.example.com
app02.example.com

[databases]
dbms01.example.com
dbms02.example.com

[centos:children]
apps
databases

[ubuntu:children]
frontends

[ubuntu:vars]
testvar=group
```

Now, let's run an ad hoc command to see what value of [testvar] is actually set:

```
cd /root/Desktop/gitlab-ci-ansible-course/Lab_15
ansible -i hostgroups-children-vars-ini ubuntu -m debug -a "var=testvar"

frt01.example.com | SUCCESS => {
    "testvar": "childgroup"
}
```

```
frt02.example.com | SUCCESS => {
    "testvar": "childgroup"
}
```

It's important to note that the [frontends] group is a child of the [ubuntu] group in this inventory and so the variable value we set at the [frontends] group level wins as this is the child group in this scenario.

Special host management using patterns

As a starting point, let's consider again an inventory that we defined earlier in this lab for the purposes of exploring host groups and child groups. For your convenience, the inventory contents are provided again here:

```
loadbalancer.example.com

[frontends]
frt01.example.com
frt02.example.com

[apps]
app01.example.com
app02.example.com

[databases]
dbms01.example.com
dbms02.example.com

[centos:children]
apps
databases

[ubuntu:children]
frontends
```

To demonstrate host/group selection by pattern, we shall use the [--list-hosts] switch with the [ansible] command to see which hosts Ansible would operate on. You are welcome to expand the example to use the [ping] module, but we'll use [--list-hosts] here in the interests of space and keeping the output concise and readable:

1. We have already mentioned the special [all] group to specify all hosts in the inventory:

```
ansible -i hostgroups-children-ini all --list-hosts

  hosts (7):
    loadbalancer.example.com
    frt01.example.com
    frt02.example.com
    app01.example.com
    app02.example.com
    dbms01.example.com
    dbms02.example.com
```

The asterisk character has the same effect as [all], but needs to be quoted in single quotes for the shell to interpret the command properly:

```
ansible -i hostgroups-children-ini '*' --list-hosts

  hosts (7):
    loadbalancer.example.com
    frt01.example.com
    frt02.example.com
    app01.example.com
    app02.example.com
    dbms01.example.com
    dbms02.example.com
```

2. Use [:] to specify a logical [OR], meaning "apply to hosts either in this group or that group," as in this example:

```
ansible -i hostgroups-children-ini frontends:apps --list-hosts

  hosts (4):
    frt01.example.com
    frt02.example.com
    app01.example.com
    app02.example.com
```

3. Use [!] to exclude a specific group---you can combine this with other characters such as [:] to show (for example) all hosts except those in the [apps] group. Again, [!] is a special character in the shell and so you must quote your pattern string in single quotes for it to work, as in this example:

```
ansible -i hostgroups-children-ini 'all:!apps' --list-hosts

  hosts (5):
    loadbalancer.example.com
    frt01.example.com
    frt02.example.com
    dbms01.example.com
    dbms02.example.com
```

4. Use [:&] to specify a logical [AND] between two groups, for example, if we want all hosts that are in the [centos] group and the [apps] group (again, you must use single quotes in the shell):

```
ansible -i hostgroups-children-ini 'centos:&apps' --list-hosts

  hosts (2):
    app01.example.com
    app02.example.com
```

5. Use [*] wildcards in a similar manner to what you would use in the shell, as in this example:

```
ansible -i hostgroups-children-ini 'db*.example.com' --list-hosts

  hosts (2):
    dbms02.example.com
    dbms01.example.com
```

Another way you can limit which hosts a command is run on is to use the [--limit] switch with Ansible. This uses exactly the same syntax and pattern notation as in the preceding but has the advantage that you can use it with the [ansible-playbook] command, where specifying a host pattern on the command line is only supported for the [ansible] command itself. Hence, for example, you could run the following:

```
ansible-playbook -i hostgroups-children-ini site.yml --limit frontends:apps


PLAY [A simple playbook for demonstrating inventory patterns] ******************

TASK [Gathering Facts] *********************************************************
ok: [frt02.example.com]
ok: [app01.example.com]
ok: [frt01.example.com]
ok: [app02.example.com]

TASK [Ping each host] **********************************************************
ok: [app01.example.com]
ok: [app02.example.com]
ok: [frt02.example.com]
ok: [frt01.example.com]

PLAY RECAP *********************************************************************
app01.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
app02.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt01.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt02.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Patterns are a very useful and important part of working with inventories, and something you will no doubt find invaluable going forward. That concludes our lab on Ansible inventories; however, it is hoped that this has given you everything you need to work confidently with Ansible inventories.

Summary

In this lab, you learned about creating simple static inventory files and adding hosts to them. We then extended this by learning how to add host groups and assign variables to hosts. We also looked at how to organize your inventories and variables when a single flat inventory file becomes too much to handle.

In the next lab, we will learn how to develop playbooks and roles to configure, deploy, and manage remote machines using Ansible.

Questions

1. How do you add the [frontends] group variables to your inventory?

A) [[frontends::]]

B) [[frontends::values]]

C) [[frontends:host:vars]]

D) [[frontends::variables]]

E) [[frontends:vars]]

2. What enables you to automate Linux tasks such as provisioning DNS, managing DHCP, updating packages, and configuration management?

A) Playbook

B) apt

C) Cobbler

D) Bash

E) Role

3. Ansible allows you to specify an inventory file location by using the [-i] option on the command line.

A) True

B) False

c