Lab 3. Defining Your Inventory

In this lab, we will cover the following topics:

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

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). You are welcome to populate this file or to provide your own inventory for each playbook run, and it is commonplace to see inventories provided alongside playbooks. After all, there's rarely a "one size fits all" playbook, and although you can subdivide your inventory with groups (more on this later), it can often be just as easy to provide a smaller static inventory file alongside a given playbook. As you will have seen in the earlier labs of this course, 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 /home/cloud-user/inventory all -m ping
```

In this lab, we will provide some examples of both INI and YAML formatted inventory files for you to consider, as you must have an awareness of both. Personally, I have worked with Ansible for many years and worked with either INI-formatted files or dynamic inventories, but they say knowledge is power and so it will do no harm to learn a little about both formats.

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=192.168.81.142 ansible_port=3333

target2.example.com ansible_port=3333 ansible_user=danieloh

target3.example.com ansible_host=192.168.81.143 ansible_port=5555
```

Hence, the preceding three hosts can be summarized as follows:

- The [target1.example.com] host should be connected to using the [192.168.81.142] IP address, on port [3333].
- The [target2.example.com] host should be connected to on port [3333] also, but this time using the [danieloh] user rather than the account running the Ansible command.
- The [target3.example.com] host should be connected to using the [192.168.81.143] IP address, on port [5555].

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:

```
---
ungrouped:
  hosts:
    target1.example.com:
```

```
      ansible_host: 192.168.81.142
      ansible_port: 3333
    target2.example.com:
      ansible_port: 3333
      ansible_user: danieloh
    target3.example.com:
      ansible_host: 192.168.81.143
      ansible_port: 5555
```

You may come across inventory file examples containing parameters such as [ansible_ssh_port], [ansible_ssh_host], and [ansible_ssh_user]--these variable names (and others like them) were used in Ansible versions before 2.0. Backward compatibility has been maintained for many of these, but you should update them where possible as this compatibility may be removed at some point in the future.

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.

Before we proceed any further, it's worth noting that inventories can also contain groups of groups, which is incredibly useful for processing certain tasks by a different division. The preceding inventory stands in its own right, but what if our frontend servers are built on Ubuntu, and the app and database servers are built on CentOS? There will be some fundamental differences in the ways we handle these hosts---for example, we might use the [apt] module to manage packages on Ubuntu and the [yum] module on CentOS.

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:
```

You can see that the [children] keyword is still used in the YAML-formatted inventory, but now the structure is more hierarchical than it was in the INI format. The indentation might be easier for you to follow, but note how the hosts are ultimately defined at quite a high level of indentation---this format could be more difficult to extend depending on your desired approach.

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

We have already touched upon host variables---we saw them earlier in this lab when we used them to override connection details such as the user account to connect with, the address to connect to, and the port to use. However, there is so much more you can do with Ansible and inventory variables, and it is important to note that they can be defined not only at the host level but also at the group level, which again provides you with some incredibly powerful ways in which to efficiently manage your infrastructure from one central 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.

Right now, our examples are small and compact and only contain a handful of groups and variables; however, when you scale this up to a full infrastructure of servers, using a single flat inventory file could, once again, become unmanageable. Luckily, Ansible also provides a solution to this. Two specially-named directories, [host_vars] and [group_vars], are automatically searched for appropriate variable content if they exist within the playbook directory. We can test this out by recreating the preceding frontend variables example using this special directory structure, rather than putting the variables into the inventory file.

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:

```
$ 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.

The files themselves are simply an adaptation of our previous ones:

```
$ cat host_vars/frt01.example.com/main.yml
---
https_port: 8444

$ cat group_vars/frontends/https_port.yml
---
https_port: 8443

$ cat group_vars/frontends/lb_vip.yml
---
lb_vip: lb.example.com
```

Even with this 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:

```
$ 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 (hence, the group definition is [[ubuntu:children]]), and so the variable value we set at the [frontends] group level wins as this is the child group in this scenario.

By now, you should have a pretty good idea of how to work with static inventory files. No look at Ansible's inventory capabilities is complete, however, without a look at dynamic inventories, and we shall do exactly this in the next section.

Generating a dynamic inventory file

In these days of cloud computing and infrastructure-as-code, the hosts you may wish to automate could change on a daily if not hourly basis! Keeping a static Ansible inventory up to date could become a full-time job, and hence, in many large-scale scenarios, it becomes unrealistic to attempt to use a static inventory on an ongoing basis.

This is where Ansible's dynamic inventory support comes in. In short, Ansible can gather its inventory data from just about any executable file (though you will find that most dynamic inventories are written in Python)---the only requirement is that the executable returns the inventory data in a specified JSON format. You are free to create your

own inventory scripts if you wish, but thankfully, many have been created already for you to use that cover a multitude of potential inventory sources including Amazon EC2, Microsoft Azure, Red Hat Satellite, LDAP directories, and many more systems.

When writing a course, it is difficult to know for certain which dynamic inventory script to use as an example as it is not a given that everyone will have an Amazon EC2 account they can freely use to test against (for example). As a result, we will use the Cobbler provisioning system by way of example, as this is freely available and easy to roll out on a CentOS system. For those interested, Cobbler is a system for dynamically provisioning and building Linux systems, and it can handle all aspects of this including DNS, DHCP, PXE booting, and so on. Hence, if you were to use this to provision virtual or physical machines in your infrastructure, it would make sense to also use this as your inventory source as Cobbler was responsible for building the systems in the first place, and so knows all of the system names.

This example will demonstrate for you the fundamentals of working with a dynamic inventory, which you can then take forward to use the dynamic inventory scripts for other systems. Let's get started with this process by first of all installing Cobbler---the process outlined here was tested on CentOS 7.8:

1. Your first task is to install the relevant Cobbler packages using [yum]. Note that, at the time of writing, the SELinux policy provided with CentOS 7 did not support Cobbler's functionality and blocks some aspects from working. Although this is not something you should do in a production environment, your simplest path to getting this demo up and running is to simply disable SELinux:

```
$ yum install -y cobbler cobbler-web
$ setenforce 0
```

2. Next, ensure that the [cobblerd] service is configured to listen on the loopback address by checking the settings in [/etc/cobbler/settings]---the relevant snippet of the file is shown here and should appear as follows:

```
# default, localhost
server: 127.0.0.1
```

This is not a public listening address, so please *do not use* [0.0.0.0]. You can also set it to the IP address of the Cobbler server.

3. With this step complete, you can start the [cobblerd] service using [systemctl]:

```
$ systemctl start cobblerd.service
$ systemctl enable cobblerd.service
$ systemctl status cobblerd.service
```

4. With the Cobbler service up and running, we'll now step through the process of adding a distribution to Cobbler to create some hosts off of. This process is fairly simple, but you do need to add a kernel file and an initial RAM disk file. The easiest source to obtain these from is your [/boot] directory, assuming you have installed Cobbler on CentOS 7. On the test system used for this demo, the following commands were used---however, you must replace the version number in the [vmlinuz] and [initramfs] filenames with the appropriate version numbers from your system's [/boot] directory:

```
$ cobbler distro add --name=CentOS --kernel=/boot/vmlinuz-3.10.0-957.el7.x86_64 --
initrd=/boot/initramfs-3.10.0-957.el7.x86_64.img

$ cobbler profile add --name=webservers --distro=CentOS
```

This definition is quite rudimentary and would not necessarily be able to produce working server images; however, it will suffice for our simple demo as we can add some systems based on this notional CentOS-based image. Note that the profile name we are creating, [webservers], will later become our inventory group name in our dynamic inventory.

5. Let's now add those systems to Cobbler. The following two commands will add two hosts called [frontend01] and [frontend02] to our Cobbler system, using the [webservers] profile we created previously:

```
$ cobbler system add --name=frontend01 --profile=webservers --dns-
name=frontend01.example.com --interface=eth0

$ cobbler system add --name=frontend02 --profile=webservers --dns-
name=frontend02.example.com --interface=eth0
```

Note that, for Ansible to work, it must be able to reach these FQDNs specified in the [--dns-name] parameter. To achieve this, I am also adding entries to [/etc/hosts] on the Cobbler system for these two machines to ensure we can reach them later. These entries can point to any two systems of your choosing as this is just a test.

At this point, you have successfully installed Cobbler, created a profile, and added two hypothetical systems to this profile. The next stage in our process is to download and configure the Ansible dynamic inventory scripts to work with these entries. To achieve this, let's get started on the process given here:

1. Download the Cobbler dynamic inventory file from the GitHub Ansible repository and the associated configuration file template. Note that most dynamic inventory scripts provided with Ansible also have a templated configuration file, which will contain parameters that you may need to set to get the dynamic inventory script working. For our simple example, we will download these files into our current working directory:

```
$ wget
https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/cobbler.py
$ wget
https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/cobbler.ini
$ chmod +x cobbler.py
```

It is important to remember to make whatever dynamic inventory script you download executable, as shown previously; if you don't do this, then Ansible won't be able to run the script even if everything else is set up perfectly.

2. Edit the [cobbler.ini] file and ensure that it points to the localhost as, for this example, we are going to run Ansible and Cobbler on the same system. In real life, you would point it at the remote URL of your Cobbler system. A snippet of the configuration file is shown here to give you an idea of what to configure:

```
[cobbler]

# Specify IP address or Hostname of the cobbler server. The default variable is here:
host = http://127.0.0.1/cobbler_api

# (Optional) With caching, you will have responses of API call with the cobbler server
quicker
cache_path = /tmp
cache_max_age = 900
```

3. You can now run an Ansible ad hoc command in the manner you are used to---the only difference this time is that you will specify the filename of the dynamic inventory script rather than the name of the static inventory file. Assuming you have set up hosts at the two addresses we entered into Cobbler earlier, your output should look something like that shown here:

```
$  ansible -i cobbler.py webservers -m ping
frontend01.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
frontend02.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

That's it! You have just implemented your first dynamic inventory in Ansible. Of course, we know that many readers won't be using Cobbler, and some of the other dynamic inventory plugins are a little more complex to get going. For example, the Amazon EC2 dynamic inventory script requires your authentication details for Amazon Web Services (or a suitable IAM account) and the installation of the Python [boto] and [boto3] libraries. How would you know to do all of this? Luckily, it is all documented in the headers of either the dynamic inventory script or the configuration file, so the most fundamental piece of advice I can give is this: whenever you download a new dynamic inventory script, be sure to check out the files themselves in your favorite editor as their requirements have most likely been documented for you.

Using multiple inventory sources in the inventory directories

So far in this course, we have been specifying our inventory file (either static or dynamic) using the [-i] switch in our Ansible commands. What might not be apparent is that you can specify the [-i] switch more than once and so use multiple inventories at the same time. This enables you to perform tasks such as running a playbook (or ad hoc command) across hosts from both static and dynamic inventories at the same time. Ansible will work out what needs to be done---static inventories should not be marked as executable and so will not be processed as such, whereas dynamic inventories will be. This small but clever trick enables you to combine multiple inventory sources with ease. Let's move on in the next section to looking at the use of static inventory groups in combination with dynamic ones, an extension of this multiple-inventory functionality.

Using static groups with dynamic groups

Of course, the possibility of mixing inventories brings with it an interesting question---what happens to the groups from a dynamic inventory and a static inventory if you define both? The answer is that Ansible combines both, and this leads to an interesting possibility. As you will have observed, our Cobbler inventory script produced an Ansible group called [webservers] from a Cobbler profile that we called [webservers]. This is common for most dynamic inventory providers; most inventory sources (for example, Cobbler and Amazon EC2) are not Ansible-aware and so do not offer groups that Ansible can directly use. As a result, most dynamic inventory scripts will use some facet of information from the inventory source to produce groupings, the Cobbler machine profile being one such example.

Let's extend our Cobbler example from the preceding section by mixing a static inventory. Suppose that we want to make our [webservers] machines a child group of a group called [centos] so that we can, in the future, group all CentOS machines together. We know that we only have a Cobbler profile called [webservers], and ideally, we don't want to start messing with the Cobbler setup to do something solely Ansible-related.

The answer to this is to create a static inventory file with two group definitions. The first must be the same name as the group you are expecting from the dynamic inventory, except that you should leave it blank. When Ansible combines the static and dynamic inventory contents, it will overlap the two groups and so add the hosts from Cobbler to these [webservers] groups.

The second group definition should state that [webservers] is a child group of the [centos] group. The resulting file should look something like this:

```
[webservers]

[centos:children]
webservers
```

Now let's run a simple ad hoc [ping] command in Ansible to see how it evaluates the two inventories together. Notice how we will specify the [centos] group to run [ping] against, instead of the [webservers] group. We know that Cobbler has no [centos] group because we never created one, and we know that any hosts in this group must come via the [webservers] group when you combine the two inventories, as our static inventory has no hosts in it. The results will look something like this:

```
$ ansible -i static-groups-mix-ini -i cobbler.py centos -m ping
frontend01.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
frontend02.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

As you can see from the preceding output, we have referenced two different inventories, one static and the other dynamic. We have combined groups, taking hosts that only exist in one inventory source, and combining them with a group that only exists in another. As you can see, this is an incredibly simple example, and it would be easy to extend this to combine lists of static and dynamic hosts or to add a custom variable to a host that comes from a dynamic inventory.

Special host management using patterns

We have already established that you will often want to run either an ad hoc command or a playbook against only a subsection of your inventory. So far, we have been quite precise in doing that, but let's now expand upon this by looking at how Ansible can work with patterns to figure out which hosts a command (or playbook) should be run against.

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. We then learned how to make use of dynamic inventory files, before concluding with a look at useful tips and tricks such as combining inventory sources and using patterns to specify hosts, all of which will make how you work with inventories easier and yet simultaneously more powerful.

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) Yum

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