Lab 2. Understanding the Fundamentals of Ansible

In this lab, we will cover the following topics:

- Getting familiar with the Ansible framework
- Exploring the configuration file
- Command-line arguments
- Defining variables
- Understanding Jinja2 filters

Getting familiar with the Ansible framework

In order to run Ansible's ad hoc commands via an SSH connection from your Ansible control machine to multiple remote hosts, you need to ensure you have the latest Ansible version installed on the control host. Use the following command to confirm the latest Ansible version:

```
$ ansible --version
ansible 2.9.6
 config file = /etc/ansible/ansible.cfg
 configured module search path = [u'/home/jamesf_local/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
 ansible python module location = /usr/lib/python2.7/dist-packages/ansible
 executable location = /usr/bin/ansible
 python version = 2.7.17 (default, Nov 7 2019, 10:07:09) [GCC 9.2.1 20191008]
```

You also need to ensure SSH connectivity with each remote host that you will define in the inventory. You can use a simple, manual SSH connection on each of your remote hosts to test the connectivity, as Ansible will make use of SSH during all remote Linux-based automation tasks:

```
$ ssh root@frontend.example.com
The authenticity of host 'frontend.example.com (192.168.1.52)' can't be established.
ED25519 key fingerprint is SHA256:hU+saFERGFDERW453tasdFPAkpVws.
Are you sure you want to continue connecting (yes/no)? yes
password:<Input_Your_Password>
```

In this section, we will walk you through how Ansible works, starting with some simple connectivity testing. You can learn how the Ansible framework accesses multiple host machines to execute your tasks by following this simple procedure:

1. Create or edit your default inventory file, [/etc/ansible/hosts] (you can also specify the path with your own inventory file by passing options such as [---inventory=/path/inventory_file]). Add some example hosts to your inventory---these must be the IP addresses or hostnames of real machines for Ansible to test against. The following are examples from my network, but you need to substitute these for your own devices. Add one hostname (or IP address) per line:

```
frontend.example.com
backend1.example.com
backend2.example.com
```

All hosts should be specified with a resolvable address---that is, a **Fully Qualified Domain Name** (**FQDN**)---if your hosts have DNS entries (or are in [/etc/hosts] on your Ansible control node). This can be IP addresses if you do not have DNS or host entries set up. Whatever format you choose for your inventory addresses, you should be able to successfully ping each host. See the following output as an example:

```
$ ping frontend.example.com

PING frontend.example.com (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost (127.0.0.1): icmp_seq=1 ttl=64 time=0.022 ms
64 bytes from localhost (127.0.0.1): icmp_seq=2 ttl=64 time=0.018 ms
64 bytes from localhost (127.0.0.1): icmp_seq=3 ttl=64 time=0.045 ms
64 bytes from localhost (127.0.0.1): icmp_seq=4 ttl=64 time=0.033 ms
```

2. To make the automation process seamless, we'll generate an SSH authentication key pair so that we don't have to type in a password every time we want to run a playbook. If you do not already have an SSH key pair, you can generate one using the following command:

```
$ ssh-keygen
```

When you run the [ssh-keygen] tool, you will see an output similar to the following. Note that you should leave the [passphrase] variable blank when prompted; otherwise, you will need to enter a passphrase every time you want to run an Ansible task, which removes the convenience of authenticating with SSH keys:

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/Users/doh/.ssh/id_rsa): <Enter>
Enter passphrase (empty for no passphrase): <Press Enter>
Enter same passphrase again: <Press Enter>
Your identification has been saved in /Users/doh/.ssh/id_rsa.
Your public key has been saved in /Users/doh/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:1IF0KMMTVAMEQF62kTwcG59okGZLiMmi4Ae/BGBT+24 doh@fenago.com
The key's randomart image is:
+---[RSA 2048]----+
|=*=*BB==+oo |
|B=*+*B=.o+ . |
|=+=o=.o+. . |
|...=. . |
| o .. S |
| .. |
| E |
| . |
| |
+----[SHA256]-----+
```

3. Although there are conditions that your SSH keys are automatically picked up with, it is recommended that you make use of [ssh-agent] as this allows you to load multiple keys to authenticate against a variety of targets. This will be very useful to you in the future, even if it isn't right now. Start [ssh-agent] and add your new authentication key, as follows (note that you will need to do this for every shell that you open):

```
$ ssh-agent bash
$ ssh-add ~/.ssh/id_rsa
```

Before you can perform key-based authentication with your target hosts, you need to apply the public key from the key pair you just generated to each host. You can copy the key to each host, in turn, using the following command:

```
$  ssh-copy-id -i ~/.ssh/id_rsa.pub frontend.example.com
/usr/bin/ssh-copy-id: INFO: Source of key(s) to be installed: "~/.ssh/id_rsa.pub"
/usr/bin/ssh-copy-id: INFO: attempting to log in with the new key(s), to filter out
any that are already installed
/usr/bin/ssh-copy-id: INFO: 1 key(s) remain to be installed -- if you are prompted now
it is to install the new keys
doh@frontend.example.com's password:

Number of key(s) added: 1

Now try logging into the machine, with: "ssh 'frontend.example.com'"
and check to make sure that only the key(s) you wanted were added.
```

With this complete, you should now be able to perform an Ansible ping command on the hosts you put in your inventory file. You will find that you are not prompted for a password at any point as the SSH connections to all the hosts in your inventory are authenticated with your SSH key pair. So, you should see an output similar to the following:

```
$ ansible all -i hosts -m ping
frontend.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
backend1.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
backend2.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

This example output is generated with Ansible's default level of verbosity. If you run into problems during this process, you can increase Ansible's level of verbosity by passing one or more [-v] switches to the [ansible] command when you run it. For most issues, it is recommended that you use [-vvvv], which gives you ample debugging information, including the raw SSH commands and the output from them. For example, assume that a certain host (such as [backend2.example.com]) can't be connected to and you receive an error similar to the following:

```
backend2.example.com | FAILED => SSH encountered an unknown error during the
connection. We recommend you re-run the command using -vvvv, which will enable SSH
debugging output to help diagnose the issue
```

`ansible all -m ping -vvvv`

Breaking down the Ansible components

In order to understand the various components, we first need an inventory to work from. Let's create an example one, ideally with multiple hosts in it---this could be the same as the one you created in the previous section. As discussed in that section, you should populate the inventory with the hostnames or IP addresses of the hosts that you can reach from the control host itself:

```
remote1.example.com
remote2.example.com
remote3.example.com
```

To really understand how Ansible---as well as its various components---works, we first need to create an Ansible playbook. While the ad hoc commands that we have experimented with so far are just single tasks, playbooks are organized groups of tasks that are (usually) run in sequence. Conditional logic can be applied and in any other programming language, they would be considered your code. At the head of the playbook, you should specify the name of your play---although this is not mandatory, it is good practice to name all your plays and tasks as without this, it would be quite hard for someone else to interpret what the playbook does, or even for you to if you come back to it after a period of time. Let's get started with building our first example playbook:

1. Specify the play name and inventory hosts to run your tasks against at the very top of your playbook. Also, note the use of [---], which denotes the beginning of a YAML file (Ansible playbooks that are written in YAML):

```
---
- name: My first Ansible playbook
  hosts: all
```

2. After this, we will tell Ansible that we want to perform all the tasks in this playbook as a superuser (usually [root]). We do this with the following statement (to aid your memory, think of [become] as shorthand for [become superuser]):

```
  become: yes
```

3. After this header, we will specify a task block that will contain one or more tasks to be run in sequence. For now, we will simply create one task to update the version of Apache using the [apt] module (because of this, this playbook is only suitable for running against RHEL-, CentOS-, or Fedora-based hosts). We will also specify a special element of the play called a handler. Simply put, a handler is a special type of task that is called only if something changes. So, in this example, it restarts the web server, but only if it changes, preventing unnecessary restarts if the playbook is run several times and there are no updates for Apache. The following code performs these functions exactly and should form the basis of your first playbook:

```
  tasks:
  - name: Update the latest of an Apache Web Server
    apt:
      name: apache2
      state: latest
    notify:
      - Restart an Apache Web Server

 handlers:
 - name: Restart an Apache Web Server
   service:
     name: apache2
     state: restarted
```

Congratulations, you now have your very first Ansible playbook! If you run this now, you should see it iterate through all the hosts in your inventory, as well as on each update in the Apache package, and then restart the service where the package was updated. Your output should look something as follows:

```
$ cd ~/Desktop/ansible-course/Lab_2
$ ansible-playbook update-apache-version.yml


$ PLAY [My first Ansible playbook] **********************************************

TASK [Gathering Facts] *********************************************************
ok: [remote2.example.com]
ok: [remote1.example.com]
ok: [remote3.example.com]

TASK [Update the latest of an Apache Web Server] *******************************
changed: [remote2.example.com]
changed: [remote3.example.com]
changed: [remote1.example.com]

RUNNING HANDLER [Restart an Apache Web Server] *********************************
changed: [remote3.example.com]
changed: [remote1.example.com]
changed: [remote2.example.com]

PLAY RECAP *********************************************************************
remote1.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
remote2.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
remote3.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Since we have added all above hosts in `/etc/hosts` with `127.0.0.1` address so they are pointing to same machine, we will get locking error which can be ignored. This error won't occur using multiple machines

If we run the playbook a second time, we know that it is very unlikely that the Apache package will need upgrading again. Notice how the playbook output differs this time:

```
PLAY [My first Ansible playbook] **********************************************

TASK [Gathering Facts] *********************************************************
ok: [remote1.example.com]
ok: [remote2.example.com]
ok: [remote3.example.com]

TASK [Update the latest of an Apache Web Server] *******************************
ok: [remote2.example.com]
ok: [remote3.example.com]
ok: [remote1.example.com]

PLAY RECAP *********************************************************************
remote1.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
remote2.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

```
remote3.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

You can see that this time, the output from the [Update the latest of an Apache Web Server] task is [ok] for all three hosts, meaning no changes were applied (the package was not updated). As a result of this, our handler is not notified and does not run---you can see that it does not even feature in the preceding playbook output. This distinction is important---the goal of an Ansible playbook (and the modules that underpin Ansible) should be to only make changes when they need to be made. If everything is all up to date, then the target host should not be altered. Unnecessary restarts to services should be avoided, as should unnecessary alterations to files. In short, Ansible playbooks are (and should be) designed to be efficient and to achieve a target machine state.

Learning the YAML syntax

In this section, you will learn how to write a YAML file with the correct syntax and best practices and tips for running a playbook on multiple remote machines. Ansible uses YAML because it is easier for humans to read and write than other common data formats, such as XML or JSON. There are no commas, curly braces, or tags to worry about, and the enforced indentation in the code ensures that it is tidy and easy on the eye. In addition, there are libraries available in most programming languages for working with YAML.

Let's explore the YAML language through the example playbook we created in the preceding section:

1. Lists are an important construct in the YAML language---in fact, although it might not be obvious, the [tasks:] block of the playbook is actually a YAML list. A list in YAML lists all of its items at the same indentation level, with each line starting with [-]. For example, we updated the [apache2] package from the preceding playbook using the following code:

```
- name: Update the latest of an Apache Web Server
  apt:
    name: apache2
    state: latest
```

However, we could have specified a list of packages to be upgraded as follows:

```
- name: Update the latest of an Apache Web Server
  apt:
    name:
      - apache2
      - mod_ssl
    state: latest
```

Now, rather than passing a single value to the [name:] key, we pass a YAML-formatted list containing the names of two packages to be updated.

2. Dictionaries are another important concept in YAML---they are represented by a [key: value] format, as we have already extensively seen, but all of the items in the dictionary are indented by one more level. This is easiest explained by an example, so consider the following code from our example playbook:

```
  service:
    name: apache2
    state: restarted
```

In this example (from [handler]), the [service] definition is actually a dictionary and both the [name] and [state] keys are indented with two more spaces than the [service] key. This higher level of indentation means that the [name] and

[state] keys are associated with the [service] key, therefore, in this case, telling the [service] module which service to operate on ([apache2]) and what to do with it (restart it).

Already, we have observed in these two examples that you can produce quite complicated data structures by mixing lists and dictionaries.

3. As you become more advanced at playbook design (we will see examples of this later on in this course), you may very well start to produce quite complicated variable structures that you will put into their own separate files to keep your playbook code readable. The following is an example of a [variables] file that provides the details of two employees of a company:

```
---
employees:
  - name: daniel
    fullname: Fenago
    role: DevOps Evangelist
    level: Expert
    skills:
      - Kubernetes
      - Microservices
      - Ansible
      - Linux Container
  - name: michael
    fullname: Michael Smiths
    role: Enterprise Architect
    level: Advanced
    skills:
      - Cloud
      - Middleware
      - Windows
      - Storage
```

In this example, you can see that we have a dictionary containing the details of each employee. The employees themselves are list items (you can spot this because the lines start with [-]) and equally, the employee skills are denoted as list items. You will notice the [fullname], [role], [level], and [skills] keys are at the same indentation level as [name] but do not feature [-] before them. This tells you that they are in the dictionary with the list item itself, and so they represent the details of the employee.

4. YAML is very literal when it comes to parsing the language and a new line always represents a new line of code. What if you actually need to add a block of text (for example, to a variable)? In this case, you can use a literal block scalar, [|], to write multiple lines and YAML will faithfully preserve the new lines, carriage returns, and all the whitespace that follows each line (note, however, that the indentation at the beginning of each line is part of the YAML syntax):

```
Specialty: |
  Agile methodology
  Cloud-native app development practices
  Advanced enterprise DevOps practices
```

So, if we were to get Ansible to print the preceding contents to the screen, it would display as follows (note that the preceding two spaces have gone---they were interpreted correctly as part of the YAML language and not printed):

```
Agile methodology
Cloud-native app development practices
Advanced enterprise DevOps practices
```

Similar to the preceding is the folded block scalar, [>], which does the same as the literal block scalar but does not preserve line endings. This is useful for very long strings that you want to print on a single line, but also want to wrap across multiple lines in your code for the purpose of readability. Take the following variation on our example:

```
Specialty: >
  Agile methodology
  Cloud-native app development practices
  Advanced enterprise DevOps practices
```

Now, if we were to print this, we would see the following:

```
Agile methodologyCloud-native app development practicesAdvanced enterprise DevOps
practices
```

We could add trailing spaces to the preceding example to stop the words from running into each other, but I have not done this here as I wanted to provide you with an easy-to-interpret example.

As you review playbooks, variable files, and so on, you will see these structures used over and over again. Although simple in definition, they are very important---a missed level of indentation or a missing [-] instance at the start of a list item can cause your entire playbook to fail to run. As we discovered, you can put all of these various constructs together. One additional example is provided in the following code block of a [variables] file for you to consider, which shows the various examples we have covered all in one place:

```
---
servers:
  - frontend
  - backend
  - database
  - cache
employees:
  - name: daniel
    fullname: Fenago
    role: DevOps Evangelist
    level: Expert
    skills:
      - Kubernetes
      - Microservices
      - Ansible
      - Linux Container
  - name: michael
    fullname: Michael Smiths
    role: Enterprise Architect
    level: Advanced
    skills:
      - Cloud
      - Middleware
      - Windows
      - Storage
    Speciality: |
```

```
      Agile methodology
      Cloud-native app development practices
      Advanced enterprise DevOps practices
```

You can also express both dictionaries and lists in an abbreviated form, known as **flow collections**. The following example shows exactly the same data structure as our original [employees] variable file:

```
---
employees: [{"fullname": "Fenago","level": "Expert","name": "daniel","role": "DevOps
Evangelist","skills": ["Kubernetes","Microservices","Ansible","Linux Container"]},
{"fullname": "Michael Smiths","level": "Advanced","name": "michael","role":
"Enterprise Architect","skills":["Cloud","Middleware","Windows","Storage"]}]
```

Although this displays exactly the same data structure, you can see how difficult it is to read with the naked eye. Flow collections are not used extensively in YAML and I would not recommend you to make use of them yourself, but it is important to understand them in case you come across them. You will also notice that although we've started talking about variables in YAML, we haven't expressed any variable types. YAML tries to make assumptions about variable types based on the data they contain, so if you want assign [1.0] to a variable, YAML will assume it is a floating-point number. If you need to express it as a string (perhaps because it is a version number), you need to put quotation marks around it, which causes the YAML parser to interpret it as a string instead, such as in the following example:

```
version: "2.0"
```

This completes our look at the YAML language syntax. Now that's complete, in the next section, let's take a look at ways that you can organize your automation code to keep it manageable and tidy.

Organizing your automation code

As you can imagine, if you were to write all of your required Ansible tasks in one massive playbook, it would quickly become unmanageable---that is to say, it would be difficult to read, difficult for someone else to pick up and understand, and---most of all---difficult to debug when things go wrong. Ansible provides a number of ways for you to divide your code into manageable chunks; perhaps the most important of these is the use of roles. Roles (for the sake of a simple analogy) behave like a library in a conventional high-level programming language.

Let's build up a practical example. To start, we know that we need to create an inventory for Ansible to run against. In this instance, we'll create four notional groups of servers, with each group containing two servers. Our hypothetical example will contain a frontend server and application servers for a fictional application, located in two different geographic locations. Our inventory file will be called [production-inventory] and the example contents are as follows:

```
[frontends_na_zone]
frontend1-na.example.com
frontend2-na.example.com

[frontends_emea_zone]
frontend1-emea.example.com
frontend2-emea.example.com

[appservers_na_zone]
appserver1-na.example.com
appserver2-na.example.com

[appservers_emea_zone]
```

```
appserver1-emea.example.com
appserver2-emea.example.com
```

Now, obviously, we could just write one massive playbook to address the required tasks on these different hosts, but as we have already discussed, this would be cumbersome and inefficient. Let's instead break the task of automating these different hosts down into smaller playbooks:

1. [Create a playbook to run a connection test on a specific host group, such as [frontends_na_zone]. Put the following contents into the playbook:]

```
---
- hosts: frontends_na_zone
  remote_user: root
  tasks:
    - name: simple connection test
      ping:
```

2. Now, try running this playbook against the hosts (note that we have configured it to connect to a remote user on the inventory system, called [fenago], so you will either need to create this user and set up the appropriate SSH keys or change the user in the [remote_user] line of your playbook). When you run the playbook after setting up the authentication, you should see an output similar to the following:

```
$ ansible-playbook -i production-inventory frontends-na.yml

PLAY [frontends_na_zone] ********************************************************

TASK [Gathering Facts] ***********************************************************
ok: [frontend1-na.example.com]
ok: [frontend2-na.example.com]

TASK [simple connection test] ****************************************************
ok: [frontend1-na.example.com]
ok: [frontend2-na.example.com]

PLAY RECAP ***********************************************************************
frontend1-na.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frontend2-na.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

3. Now, let's extend our simple example by creating a playbook that will only run on the application servers. Again, we will use the Ansible [ping] module to perform a connection test, but in a real-world situation, you would perform more complex tasks, such as installing packages or modifying files. Specify that this playbook is run against this host group from the [[appservers_emea_zone] inventory. Add the following contents to the playbook:]

```
---
- hosts: appservers_emea_zone
  remote_user: root
  tasks:
    - name: simple connection test
      ping:
```

As before, you need to ensure you can access these servers, so either create the [fenago] user and set up authentication to that account or change the [remote_user] line in the example playbook. Once you have done this, you should be able to run the playbook and you will see an output similar to the following:

```
$ ansible-playbook -i production-inventory appservers-emea.yml

PLAY [appservers_emea_zone] ****************************************************

TASK [Gathering Facts] *********************************************************
ok: [appserver2-emea.example.com]
ok: [appserver1-emea.example.com]

TASK [simple connection test] **************************************************
ok: [appserver2-emea.example.com]
ok: [appserver1-emea.example.com]

PLAY RECAP *********************************************************************
appserver1-emea.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
appserver2-emea.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
```

4. So far, so good. However, we now have two playbooks that we need to run manually, which only addresses two of our inventory host groups. If we want to address all four groups, we need to create a total of four playbooks, all of which need to be run manually. This is hardly reflective of best automation practices. What if there was a way to take these individual playbooks and run them together from one top-level playbook? This would enable us to divide our code to keep it manageable, but also prevents a lot of manual effort when it comes to running the playbooks. Fortunately, we can do exactly that by taking advantage of the [import_playbook] directive in a top-level playbook that we will call [site.yml]:

```
---
- import_playbook: frontend-na.yml
- import_playbook: appserver-emea.yml
```

Now, when you run this single playbook using the (by now, familiar) [ansible-playbook] command, you will see that the effect is the same as if we had actually run both playbooks back to back. In this way, even before we explore the concept of roles, you can see that Ansible supports splitting up your code into manageable chunks without needing to run each chunk manually:

```
$ ansible-playbook -i production-inventory site.yml

PLAY [frontends_na_zone] *******************************************************

TASK [Gathering Facts] *********************************************************
ok: [frontend2-na.example.com]
ok: [frontend1-na.example.com]

TASK [simple connection test] **************************************************
ok: [frontend1-na.example.com]
ok: [frontend2-na.example.com]

PLAY [appservers_emea_zone] ****************************************************
```

```
TASK [Gathering Facts] *********************************************************
ok: [appserver2-emea.example.com]
ok: [appserver1-emea.example.com]

TASK [simple connection test] **************************************************
ok: [appserver2-emea.example.com]
ok: [appserver1-emea.example.com]

PLAY RECAP *********************************************************************
appserver1-emea.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
appserver2-emea.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0
rescued=0 ignored=0
frontend1-na.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frontend2-na.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

In the next section, with a look at the configuration file and some of the key directives that you might find valuable.

Exploring the configuration file

Ansible's behavior is, in part, defined by its configuration file. The central configuration file (which impacts the behavior of Ansible for all users on the system) can be found at [/etc/ansible/ansible.cfg]. However, this is not the only place Ansible will look for its configuration; in fact, it will look in the following locations, from the top to the bottom.

The first instance of the file is the configuration it will use; all of the others are ignored, even if they are present:

1. [ANSIBLE_CONFIG]: The file location specified by the value of this environment variable, if set
2. [ansible.cfg]: In the current working directory
3. [~/.ansible.cfg]: In the home directory of the user
4. [/etc/ansible/ansible.cfg]: The central configuration that we previously mentioned

If you installed Ansible through a package manager, such as [apt] or [apt], you will almost always find a default configuration file called [ansible.cfg] in [/etc/ansible]. However, if you built Ansible from the source or installed it via [pip], the central configuration file will not exist and you will need to create it yourself. A good starting point is to reference the example Ansible configuration file that is included with the source code, a copy of which can be found on GitHub at https://raw.githubusercontent.com/ansible/ansible/devel/examples/ansible.cfg.

In this section, we will detail how to locate Ansible's running configuration and how to manipulate it. Most people who install Ansible through a package find that they can get a long way with Ansible before they have to modify the default configuration, as it has been carefully designed to work in a great many scenarios. However, it is important to know a little about configuring Ansible in case you come across an issue in your environment that can only be changed by modifying the configuration.

Obviously, if you don't have Ansible installed, there's little point in exploring its configuration, so let's just check whether you have Ansible installed and working by issuing a command such as the following (the output shown is from the latest version of Ansible at the time of writing, installed on macOS with Homebrew):

```
$ ansible 2.9.6
  config file = None
  configured module search path = ['/Users/james/.ansible/plugins/modules',
```

```
'/usr/share/ansible/plugins/modules']
  ansible python module location =
/usr/local/Cellar/ansible/2.9.6_1/libexec/lib/python3.8/site-packages/ansible
  executable location = /usr/local/bin/ansible
  python version = 3.8.2 (default, Mar 11 2020, 00:28:52) [Clang 11.0.0 (clang-
1100.0.33.17)]
```

Let's get started by exploring the default configuration that is provided with Ansible:

1. The command in the following code block lists the current configuration parameters supported by Ansible. It is incredibly useful because it tells you both the environment variable that can be used to change the setting (see the [env] field) as well as the configuration file parameter and section that can be used (see the [ini] field). Other valuable information, including the default configuration values and a description of the configuration, is given (see the [default] and [description] fields, respectively). All of the information is sourced from [lib/constants.py]. Run the following command to explore the output:

```
$ ansible-config list
```

The following is an example of the kind of output you will see. There are, of course, many pages to it, but a snippet is shown here as an example:

```
$ ansible-config list
ACTION_WARNINGS:
  default: true
  description:
  - By default Ansible will issue a warning when received from a task action (module
    or action plugin)
  - These warnings can be silenced by adjusting this setting to False.
  env:
  - name: ANSIBLE_ACTION_WARNINGS
  ini:
  - key: action_warnings
    section: defaults
  name: Toggle action warnings
  type: boolean
  version_added: '2.5'
AGNOSTIC_BECOME_PROMPT:
  default: true
  description: Display an agnostic become prompt instead of displaying a prompt
containing
    the command line supplied become method
  env:
  - name: ANSIBLE_AGNOSTIC_BECOME_PROMPT
  ini:
  - key: agnostic_become_prompt
    section: privilege_escalation
  name: Display an agnostic become prompt
  type: boolean
  version_added: '2.5'
  yaml:
    key: privilege_escalation.agnostic_become_prompt
.....
```

2. If you want to see a straightforward display of all the possible configuration parameters, along with their current values (regardless of whether they are configured from environment variables or a configuration file in one of the previously listed locations), you can run the following command:

```
$ ansible-config dump
```

The output shows all the configuration parameters (in an environment variable format), along with the current settings. If the parameter is configured with its default value, you are told so (see the [(default)] element after each parameter name):

```
$ ansible-config dump
ACTION_WARNINGS(default) = True
AGNOSTIC_BECOME_PROMPT(default) = True
ALLOW_WORLD_READABLE_TMPFILES(default) = False
ANSIBLE_CONNECTION_PATH(default) = None
ANSIBLE_COW_PATH(default) = None
ANSIBLE_COW_SELECTION(default) = default
ANSIBLE_COW_WHITELIST(default) = ['bud-frogs', 'bunny', 'cheese', 'daemon', 'default',
'dragon', 'elephant-in-snake', 'elephant', 'eyes', 'hellokitty', 'kitty', 'luke-
koala', 'meow', 'milk', 'moofasa', 'moose', 'ren', 'sheep', 'small', 'stegosaurus',
'stimpy', 'supermilker', 'three-eyes', 'turkey', 'turtle', 'tux', 'udder', 'vader-
koala', 'vader', 'www']
ANSIBLE_FORCE_COLOR(default) = False
ANSIBLE_NOCOLOR(default) = False
ANSIBLE_NOCOWS(default) = False
ANSIBLE_PIPELINING(default) = False
ANSIBLE_SSH_ARGS(default) = -C -o ControlMaster=auto -o ControlPersist=60s
ANSIBLE_SSH_CONTROL_PATH(default) = None
ANSIBLE_SSH_CONTROL_PATH_DIR(default) = ~/.ansible/cp
....
```

3. Let's see the effect on this output by editing one of the configuration parameters. Let's do this by setting an environment variable, as follows (this command has been tested in the [bash] shell, but may differ for other shells):

```
$ export ANSIBLE_FORCE_COLOR=True
```

Now, let's re-run the [ansible-config] command, but this time get it to tell us only the parameters that have been changed from their default values:

```
$ ansible-config dump --only-change
ANSIBLE_FORCE_COLOR(env: ANSIBLE_FORCE_COLOR) = True
```

Here, you can see that [ansible-config] tells us that we have only changed [ANSIBLE_FORCE_COLOR] from the default value, that it is set to [True], and that we set it through an [env] variable. This is incredibly valuable, especially if you have to debug configuration issues.

When working with the Ansible configuration file itself, you will note that it is in INI format, meaning it has sections such as [[defaults]], parameters in the format [key = value], and comments beginning with either [#] or [;]. You only need to place the parameters you wish to change from their defaults in your configuration file, so if you wanted to create a simple configuration to change the location of your default inventory file, it might look as follows:

```
# Set my configuration variables
[defaults]
inventory = /Users/fenago/ansible/hosts ; Here is the path of the inventory file
```

As discussed earlier, one of the possible valid locations for the [ansible.cfg] configuration file is in your current working directory. It is likely that this is within your home directory, so on a multi-user system, we strongly recommend you restrict access to the Ansible configuration file to your user account alone. You should take all the usual precautions when it comes to securing important configuration files on a multi-user system, especially as Ansible is normally used to configure multiple remote systems and so, a lot of damage could be done if a configuration file is inadvertently compromised!

Of course, Ansible's behavior is not just controlled by the configuration files and switches---the command-line arguments that you pass to the various Ansible executables are also of vital importance. In fact, we have already worked with one already---in the preceding example, we showed you how to change where Ansible looks for its inventory file using the [inventory] parameter in [ansible.cfg]. However, in many of the examples that we previously covered in this course, we overrode this with the [-i] switch when running Ansible. So, let's proceed to the next section to look at the use of command-line arguments when running Ansible.

Command-line arguments

In this section, you will learn about the use of command-line arguments for playbook execution and how to employ some of the more commonly used ones to your advantage. We are already very familiar with one of these arguments, the [--version] switch, which we use to confirm that Ansible is installed (and which version is installed):

```
$ ansible 2.9.6
  config file = None
  configured module search path = ['/Users/james/.ansible/plugins/modules',
'/usr/share/ansible/plugins/modules']
  ansible python module location =
/usr/local/Cellar/ansible/2.9.6_1/libexec/lib/python3.8/site-packages/ansible
  executable location = /usr/local/bin/ansible
  python version = 3.8.2 (default, Mar 11 2020, 00:28:52) [Clang 11.0.0 (clang-
1100.0.33.17)]
```

Just as we were able to learn about the various configuration parameters directly through Ansible, we can also learn about the command-line arguments. Almost all of the Ansible executables have a [--help] option that you can run to display the valid command-line parameters. Let's try this out now:

1. You can view all the options and arguments when you execute the [ansible] command line. Use the following command:

```
$ ansible --help
```

2. We could take one example from the preceding code to build on our previous use of [ansible]; so far, we have almost exclusively used it to run ad hoc tasks with the [-m] and [-a] parameters. However, [ansible] can also perform useful tasks such as telling us about the hosts in a group within our inventory. We could explore this using the [production-inventory] file we used earlier in this lab:

```
$ ansible -i production-inventory --list-host appservers_emea_zone
```

When you run this, you should see the members of the [appservers_emea_zone] inventory group listed. Although perhaps a little contrived, this example is incredibly valuable when you start working with dynamic inventory files and

you can no longer just [cat] your inventory file to the terminal to view the contents:

```
$ ansible -i production-inventory --list-host appservers_emea_zone
  hosts (2):
    appserver1-emea.example.com
    appserver2-emea.example.com
```

The same is true for the [ansible-playbook] executable file, too. We have already seen a few of these in the previous examples of this course and there's more that we can do. For example, earlier, we discussed the use of [ssh-agent] to manage multiple SSH authentication keys. While this makes running playbooks simple (as you don't have to pass any authentication parameters to Ansible), it is not the only way of doing this. You can use one of the command-line arguments for [ansible-playbook] to specify the private SSH key file, instead, as follows:

```
$ ansible-playbook -i production-inventory site.yml --private-key ~/keys/id_rsa
```

Similarly, in the preceding section, we specified the [remote_user] variable for Ansible to connect with in the playbook. However, command-line arguments can also set this parameter for the playbook; so, rather than editing the [remote_user] line in the playbook, we could remove it altogether and instead have run it using the following command-line string:

```
$ ansible-playbook -i production-inventory site.yml --user root
```

Understanding ad hoc commands

We have already seen a handful of ad hoc commands so far in this course, but to recap, they are single commands you can run with Ansible, making use of Ansible modules without the need to create or save playbooks. They are very useful for performing quick, one-off tasks on a number of remote machines or for testing and understanding the behavior of the Ansible modules that you intend to use in your playbooks. They are both a great learning tool and a quick and dirty (because you never document your work with a playbook!) automation solution.

As with every Ansible example, we need an inventory to run against. Let's reuse our [production-inventory] file from before:

```
[frontends_na_zone]
frontend1-na.example.com
frontend2-na.example.com

[frontends_emea_zone]
frontend1-emea.example.com
frontend2-emea.example.com

[appservers_na_zone]
appserver1-na.example.com
appserver2-na.example.com


[appservers_emea_zone]
appserver1-emea.example.com
appserver2-emea.example.com
```

Now, let's start with perhaps the quickest and dirtiest of ad hoc commands---running a raw shell command on a group of remote machines. Suppose that you want to check that the date and time of all the frontend servers in

EMEA are in sync---you could do this by using a monitoring tool or by manually logging into each server in turn and checking the date and time. However, you can also use an Ansible ad hoc command:

1. Run the following ad hoc command to retrieve the current date and time from all of the [frontends_emea_zone] servers:

```
$ ansible -i production-inventory frontends_emea_zone -a /usr/bin/date
```

You will see that Ansible faithfully logs in to each machine in turn and runs the [date] command, returning the current date and time. Your output will look something as follows:

```
$ ansible -i production-inventory frontends_emea_zone -a /usr/bin/date
frontend1-emea.example.com | CHANGED | rc=0 >>
Sun 5 Apr 18:55:30 BST 2020
frontend2-emea.example.com | CHANGED | rc=0 >>
Sun 5 Apr 18:55:30 BST 2020
```

2. This command is run with the user account you are logged in to when the command is run. You can use a command-line argument (discussed in the previous section) to run as a different user:

```
$ ansible -i production-inventory frontends_emea_zone -a /usr/sbin/pvs -u fenago

frontend2-emea.example.com | FAILED | rc=5 >>
  WARNING: Running as a non-root user. Functionality may be unavailable.
  /run/lvm/lvmetad.socket: access failed: Permission denied
  WARNING: Failed to connect to lvmetad. Falling back to device scanning.
  /run/lock/lvm/P_global:aux: open failed: Permission denied
  Unable to obtain global lock.non-zero return code
frontend1-emea.example.com | FAILED | rc=5 >>
  WARNING: Running as a non-root user. Functionality may be unavailable.
  /run/lvm/lvmetad.socket: access failed: Permission denied
  WARNING: Failed to connect to lvmetad. Falling back to device scanning.
  /run/lock/lvm/P_global:aux: open failed: Permission denied
  Unable to obtain global lock.non-zero return code
```

3. Here, we can see that the [fenago] user account does not have the privileges required to successfully run the [pvs] command. However, we can fix this by adding the [--become] command-line argument, which tells Ansible to become [root] on the remote systems:

```
$ ansible -i production-inventory frontends_emea_zone -a /usr/sbin/pvs -u fenago --
become

frontend2-emea.example.com | FAILED | rc=-1 >>
Missing sudo password
frontend1-emea.example.com | FAILED | rc=-1 >>
Missing sudo password
```

4. We can see that the command still fails because although [fenago] is in [/etc/sudoers], it is not allowed to run commands as [root] without entering a [sudo] password. Luckily, there's a switch to get Ansible to prompt us for this at run time, meaning we don't need to edit our [/etc/sudoers] file:

```
$ ansible -i production-inventory frontends_emea_zone -a /usr/sbin/pvs -u fenago --
become --ask-become-pass
```

```
BECOME password:

frontend1-emea.example.com | CHANGED | rc=0 >>
  PV VG Fmt Attr PSize PFree
  /dev/sda2 centos lvm2 a-- <19.00g 0
frontend2-emea.example.com | CHANGED | rc=0 >>
  PV VG Fmt Attr PSize PFree
  /dev/sda2 centos lvm2 a-- <19.00g 0
```

5. By default, if you don't specify a module using the [-m] command-line argument, Ansible assumes you want to use the [command] module (see https://docs.ansible.com/ansible/latest/modules/command_module.html). If you wish to use a specific module, you can add the [-m] switch to the command-line arguments and then specify the module arguments under the [-a] switch, as in the following example:

```
$ ansible -i production-inventory frontends_emea_zone -m copy -a "src=/etc/hosts
dest=/root/Desktop/hosts"
frontend1-emea.example.com | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": true,
    "checksum": "e0637e631f4ab0aaebef1a6b8822a36f031f332e",
    "dest": "/root/Desktop/hosts",
    "gid": 0,
    "group": "root",
    "md5sum": "a7dc0d7b8902e9c8c096c93eb431d19e",
    "mode": "0644",
    "owner": "root",
    "size": 970,
    "src": "/root/.ansible/tmp/ansible-tmp-1586110004.75-208447517347027/source",
    "state": "file",
    "uid": 0
}
frontend2-emea.example.com | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": true,
    "checksum": "e0637e631f4ab0aaebef1a6b8822a36f031f332e",
    "dest": "/root/Desktop/hosts",
    "gid": 0,
    "group": "root",
    "md5sum": "a7dc0d7b8902e9c8c096c93eb431d19e",
    "mode": "0644",
    "owner": "root",
    "size": 970,
    "src": "/root/.ansible/tmp/ansible-tmp-1586110004.75-208447517347027/source",
    "state": "file",
    "uid": 0
}
```

The preceding output not only shows that the copy was performed successfully to both hosts but also all the output values from the [copy] module. This, again, can be very helpful later when you are developing playbooks as it enables you to understand exactly how the module works and what output it produces in cases where you need to perform further work with that output. This is a more advanced topic, however, that is beyond the scope of this introductory lab.

All examples we have performed so far are very quick to execute and run, but this is not always the case with computing tasks. When you have to run an operation for a long time, say more than two hours, you should consider running it as a background process. In this instance, you can run the command asynchronously and confirm the result of that execution later.

For example, to execute [sleep 2h] asynchronously in the background with a timeout of 7,200 seconds ([-B]) and without polling ([-P]), use this command:

```
$ ansible -i production-inventory frontends_emea_zone -B 7200 -P 0 -a "sleep 2h"
frontend1-emea.example.com | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "ansible_job_id": "537978889103.8857",
    "changed": true,
    "finished": 0,
    "results_file": "/root/.ansible_async/537978889103.8857",
    "started": 1
}
frontend2-emea.example.com | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "ansible_job_id": "651461662130.8858",
    "changed": true,
    "finished": 0,
    "results_file": "/root/.ansible_async/651461662130.8858",
    "started": 1
}
```

Note that the output from this command gives a unique job ID for each task on each host. Let's now say that we want to see how this task proceeds on the second frontend server. Simply issue the following command from your Ansible control machine:

```
$ ansible -i production-inventory frontend2-emea.example.com -m async_status -a
"jid=651461662130.8858"
frontend2-emea.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "ansible_job_id": "651461662130.8858",
    "changed": false,
    "finished": 0,
    "started": 1
}
```

Here, we can see that the job has started but not finished. If we now kill the [sleep] command that we issued and check on the status again, we can see the following:

```
$ ansible -i production-inventory frontend2-emea.example.com -m async_status -a
"jid=651461662130.8858"
frontend2-emea.example.com | FAILED! => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "ansible_job_id": "651461662130.8858",
    "changed": true,
    "cmd": [
        "sleep",
        "2h"
    ],
    "delta": "0:03:16.534212",
    "end": "2020-04-05 19:18:08.431258",
    "finished": 1,
    "msg": "non-zero return code",
    "rc": -15,
    "start": "2020-04-05 19:14:51.897046",
    "stderr": "",
    "stderr_lines": [],
    "stdout": "",
    "stdout_lines": []
}
```

Here, we see a [FAILED] status result because the [sleep] command was killed; it did not exit cleanly and returned a [-15] code (see the [rc] parameter). When it was killed, no output was sent to either [stdout] or [stderr], but if it had been, Ansible would have captured it and displayed it in the preceding code, which would aid you in debugging the failure. Lots of other useful information is included, including how long the task actually ran for, the end time, and so on. Similarly, the useful output is returned when the task exits cleanly.

Defining variables

Let's get started with a practical look at defining variables in Ansible.

Variables in Ansible should have well-formatted names that adhere to the following rules:

- The name of the variable must only include letters, underscores, and numbers---spaces are not allowed.
- The name of the variable can only begin with a letter---they can contain numbers, but cannot start with one.

For example, the following are good variable names:

- [external_svc_port]
- [internal_hostname_ap1]

The following examples are all invalid, however, and cannot be used:

- [appserver-zone-na]
- [cache server ip]
- [dbms.server.port]
- [01appserver]

As discussed in the *Learning the YAML syntax* section, variables can be defined in a dictionary structure, such as the following. All values are declared in key-value pairs:

```
region:
  east: app
  west: frontend
  central: cache
```

In order to retrieve a specific field from the preceding dictionary structure, you can use either one of the following notations:

```
# bracket notation
region['east']

# dot notation
region.east
```

There are some exceptions to this; for example, you should use [bracket notation] if the variable name starts and ends with two underscores (for example, [__variable__]) or contains known public attributes, such as the following:

- [as_integer_ratio]
- [symmetric_difference]

You can find more information on this at https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#creating-valid-variable-names.

This dictionary structure is valuable when defining host variables; although earlier in this lab we worked with a fictional set of employee records defined as an Ansible [variables] file, you could use this to specify something, such as some [redis] server parameters:

```
---
redis:
  - server: cacheserver01.example.com
    port: 6379
    slaveof: cacheserver02.example.com
```

These could then be applied through your playbook and one common playbook could be used for all [redis] servers, regardless of their configuration, as changeable parameters such as the [port] and [master] servers are all contained in the variables.

You can also pass set variables directly in a playbook, and even pass them to roles that you call. For example, the following playbook code calls four hypothetical roles and each assigns a different value to the [username] variable for each one. These roles could be used to set up various administration roles on a server (or multiple servers), with each passing a changing list of usernames as people come and go from the company:

```
roles:
  - role: dbms_admin
    vars:
      username: James
  - role: system_admin
    vars:
      username: John
  - role: security_amdin
    vars:
```

```
      username: Rock
  - role: app_admin
    vars:
      username: Daniel
```

To access variables from within a playbook, you simply place the variable name inside quoted pairs of curly braces. Consider the following example playbook (based loosely on our previous [redis] example):

```
---
- name: Display redis variables
  hosts: all

  vars:
    redis:
      server: cacheserver01.example.com
      port: 6379
      slaveof: cacheserver02.example.com

  tasks:
    - name: Display the redis port
      debug:
        msg: "The redis port for {{ redis.server }} is {{ redis.port }}"
```

Here, we define a variable in the playbook itself called [redis]. This variable is a dictionary, containing a number of parameters that might be important for our server. To access the contents of these variables, we use pairs of curly braces around them (as described previously) and the entire string is encased in quotation marks, which means we don't have to individually quote the variables. If you run the playbook on a local machine, you should see an output that looks as follows:

```
$ ansible-playbook -i localhost, redis-playbook.yml

PLAY [Display redis variables] *************************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [Display the redis port] **************************************************
ok: [localhost] => {
    "msg": "The redis port for cacheserver01.example.com is 6379"
}

PLAY RECAP *********************************************************************
localhost : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

Although we are accessing these variables here to print them in a debug message, you could use the same curly brace notation to assign them to module parameters, or for any other purpose that your playbook requires them for.

Understanding Jinja2 filters

Let's explore through a practical example. Suppose we have a YAML file containing some data that we want to parse. We can quite easily read a file from the machine filesystem and capture the result using the [register] keyword

([register] captures the result of the task and stores it in a variable---in the case of running the [shell] module, it captures all the output from the command that was run).

Our YAML data file might look as follows:

```
tags:
  - key: job
    value: developer
  - key: language
    value: java
```

Now, we could create a playbook to read this file and register the result, but how can we actually turn it into a variable structure that Ansible can understand and work with? Let's consider the following playbook:

```
---
- name: Jinja2 filtering demo 1
  hosts: localhost

  tasks:
    - copy:
        src: multiple-document-strings.yaml
        dest: /tmp/multiple-document-strings.yaml
    - shell: cat /tmp/multiple-document-strings.yaml
      register: result
    - debug:
        msg: '{{ item }}'
      loop: '{{ result.stdout | from_yaml_all | list }}'
```

The [shell] module does not necessarily run from the directory that the playbook is stored in, so we cannot guarantee that it will find our [multiple-document-strings.yaml] file. The [copy] module does, however, source the file from the current directory, so it is useful to use it to copy it to a known location (such as [/tmp]) for the [shell] module to read the file from. The [debug] module is then run in a [loop] module. The [loop] module is used to iterate over all of the lines of [stdout] from the [shell] command, as we are using two Jinja2 filters---[from_yaml_all] and [list].

The [from_yaml_all] filter parses the source document lines as YAML and then the [list] filter converts the parsed data into a valid Ansible list. If we run the playbook, we should see Ansible's representation of the data structure from within our original file:

```
$ ansible-playbook -i localhost, jinja2-filtering1.yml

PLAY [Jinja2 filtering demo 1] *************************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [copy] ********************************************************************
ok: [localhost]

TASK [shell] *******************************************************************
changed: [localhost]

TASK [debug] *******************************************************************
ok: [localhost] => (item={'tags': [{'value': u'developer', 'key': u'job'}, {'value':
```

```
u'java', 'key': u'language'}]}) => {
    "msg": {
        "tags": [
            {
                "key": "job",
                "value": "developer"
            },
            {
                "key": "language",
                "value": "java"
            }
        ]
    }
}

PLAY RECAP *********************************************************************
localhost : ok=4 changed=1 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

As you can see, we have generated a list of dictionaries that in themselves contain the key-value pairs.

If this data structure was already stored in our playbook, we could take this one step further and use the [items2dict] filter to turn the list into true [key: value] pairs, removing the [key] and [value] items from the data structure. For example, consider this second playbook:

```
---
- name: Jinja2 filtering demo 2
  hosts: localhost
  vars:
    tags:
      - key: job
        value: developer
      - key: language
        value: java

  tasks:
    - debug:
        msg: '{{ tags | items2dict }}'
```

Now, if we run this, we can see that our data is converted into a nice neat set of [key: value] pairs:

```
$ ansible-playbook -i localhost, jinja2-filtering2.yml


[WARNING]: Found variable using reserved name: tags

PLAY [Jinja2 filtering demo 2] *************************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [debug] *******************************************************************
ok: [localhost] => {
    "msg": {
```

```
        "job": "developer",
---
        "language": "java"
    }
}

PLAY RECAP *********************************************************************
localhost : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

Observe the warning at the top of the playbook. Ansible displays a warning if you attempt to use a reserved name for a variable, as we did here. Normally, you should not create a variable with a reserved name, but the example here demonstrates both how the filter works and how Ansible will attempt to warn you if you do something that might cause problems.

Earlier in this section, we used the [shell] module to read a file and used [register] to store the result in a variable. This is perfectly fine, if a little inelegant. Jinja2 contains a series of [lookup] filters that, among other things, can read the contents of a given file. Let's examine the behavior of this following playbook::

```
---
- name: Jinja2 filtering demo 3
  hosts: localhost
  vars:
    ping_value: "{{ lookup('file', '/etc/hosts') }}"
  tasks:
    - debug:
        msg: "ping value is {{ ping_value }}"
```

When we run this, we can see that Ansible has captured the contents of the [/etc/hosts] file for us, without us needing to resort to the [copy] and [shell] modules as we did earlier:

```
$ ansible-playbook -i localhost, jinja2-filtering3.yml

PLAY [Jinja2 filtering demo 3] *************************************************

TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [debug] *******************************************************************
ok: [localhost] => {
    "msg": "ping value is 127.0.0.1 localhost localhost.localdomain localhost4
localhost4.localdomain4\n::1 localhost localhost.localdomain localhost6
localhost6.localdomain6\n\n"
}

PLAY RECAP *********************************************************************
localhost : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

There are many other filters that you might be interested in exploring and a full list can be found in the official Jinja2 documentation (https://jinja.palletsprojects.com/en/2.11.x/). The following are a handful of other examples that will give you an idea of the kinds of things that Jinja2 filters can achieve for you, from quoting strings to concatenating lists to obtaining useful path information for a file:

```
# Add some quotation in the shell
- shell: echo {{ string_value | quote }}

# Concatenate a list into a specific string
{{ list | join("$") }}

# Have the last name of a specific file path
{{ path | basename }}

# Have the directory from a specific path
{{ path | dirname }}

# Have the directory from a specific windows path
{{ path | win_dirname }}
```

That concludes our look at Jinja2 filtering. It is a massive topic that deserves a course all to itself, but, as ever, I hope that this practical guide has given you some pointers on how to get started and where to find information.

Summary

In this lab, you learned the fundamentals of working with various Ansible programs. You then learned about the YAML syntax and the ways that you can break down your code into manageable chunks to make it easier to read and maintain. We explored the use of ad hoc commands in Ansible, variable definition and structure, and how to make use of Jinja2 filters to manipulate the data in your playbooks.

In the next lab, we will take a more in-depth look at Ansible inventories and explore some of the more advanced concepts of working with them that you may find useful.

Questions

1. Which component of Ansible allows you to define a block to execute task groups as a play?

A) [handler]

B) [service]

C) [hosts]

D) [tasks]

E) [name]

2. Which basic syntax from the YAML format do you use to start a file?

A) [###]

B) [---]

C) [%%%]

D) [===]

E) [***]

3. True or false -- in order to interpret and transform output data in Ansible, you need to use Jinja2 templates.

A) True

B) False