Lab 7. Coding Best Practices

In this lab, we will cover the following topics:

- The preferred directory layout
- The best approach to cloud inventories
- Differentiating between different environment types
- The proper approach to defining group and host variables
- Using top-level playbooks
- Leveraging version control tools
- Setting OS and distribution variances
- Porting between Ansible versions

The preferred directory layout

In this lab, let's get started with a practical example of this to show you a great way of setting up your directory structure for a simple role-based playbook that has two different inventories---one for a development environment and one for a production environment (you would want to keep these separate in any real-world use case, although ideally, you should be able to execute the same plays on both for consistency and for testing purposes).

Let's get started by building the directory structure:

1. Create a directory tree for your development inventory with the following commands:

```
$ mkdir -p inventories/development/group_vars
$ mkdir -p inventories/development/host_vars
```

2. Next, we'll define an INI-formatted inventory file for our development inventory---in our example, we'll keep this really simple with just two servers. The file to create is [inventories/development/hosts]:

```
[app]
app01.dev.example.com
app02.dev.example.com
```

3. To further our example, we'll add a group variable to our app group. Create a file called [app.yml] in the [group_vars] directory we created in the previous step:

```
---
http_port: 8080
```

4. Next, create a [production] directory structure using the same method:

```
$ mkdir -p inventories/production/group_vars
$ mkdir -p inventories/production/host_vars
```

5. Create an inventory file called [hosts] in the newly created [production] directory with the following contents:

```
[app]
app01.prod.example.com
app02.prod.example.com
```

6. Now, we'll define a different value to the [http_port] group variable for our production inventory. Add the following contents to [inventories/production/group_vars/app.yml]:

```
---
http_port: 80
```

That completes our inventory definition. Next, we will add in any custom modules or plugins that we might find useful for our playbook. Suppose we want to use the [remote_filecopy.py] module we created in Lab 5. Just as we discussed in this lab, we first create the directory for this module:

```
$ mkdir library
```

Then, add the [remote_filecopy.py] module to this library. We won't relist the code here to save space, but you can copy it from the section called *Developing custom modules* from Lab 5, or take advantage of the example code that accompanies this course on GitHub.

The same can be done for the plugins; if we also want to use our [filter] plugin that we created in Lab 6, we would create an appropriately named directory:

```
$ mkdir filter_plugins
```

Then, copy the [filter] plugin code into this directory.

Finally, we'll create a role to use in our new playbook structure. Naturally, you will have many roles, but we'll create one as an example and then you can repeat the process for each role. We'll call our role [installapp] and use the [ansible-galaxy] command (covered in Lab 4 to create the directory structure for us:

```
$ mkdir roles
$ ansible-galaxy role init --init-path roles/ installapp
- Role installapp was created successfully
```

Then, in our [roles/installapp/tasks/main.yml] file, we'll add the following contents:

```
---
- name: Display http_port variable contents
  debug:
    var: http_port

- name: Create /tmp/foo
  file:
    path: /tmp/foo
    state: file

- name: Use custom module to copy /tmp/foo
  remote_filecopy:
    source: /tmp/foo
    dest: /tmp/bar

- name: Define a fact about automation
  set_fact:
    about_automation: "Puppet is an excellent automation tool"

- name: Tell us about automation with a custom filter applied
```

```
  debug:
    msg: "{{ about_automation | improve_automation }}"
```

In the preceding code, we've reused a number of examples from earlier labs of this course. You can also define the handlers, variables, default values, and so on to the role, as discussed previously, but for our example, this will suffice.

The final stage in creating our best practice directory structure is to add a top-level playbook to run. By convention, this will be called [site.yml] and it will have the following simple contents (note that the directory structure we have built takes care of many things, allowing the top-level playbook to be incredibly simple):

```
---
- name: Play using best practise directory structure
  hosts: all

  roles:
    - installapp
```

For the purpose of clarity, your resulting directory structure should look as follows:

```
.
├── filter_plugins
│   ├── custom_filter.py
│   └── custom_filter.pyc
├── inventories
│   ├── development
│   │   ├── group_vars
│   │   │   └── app.yml
│   │   ├── hosts
│   │   └── host_vars
│   └── production
│   ├── group_vars
│   │   └── app.yml
│   ├── hosts
│   └── host_vars
├── library
│   └── remote_filecopy.py
├── roles
│   └── installapp
│   ├── defaults
│   │   └── main.yml
│   ├── files
│   ├── handlers
│   │   └── main.yml
│   ├── meta
│   │   └── main.yml
│   ├── README.md
│   ├── tasks
│   │   └── main.yml
│   ├── templates
│   ├── tests
│   │   ├── inventory
│   │   └── test.yml
│   └── vars
│   └── main.yml
└── site.yml
```

Now, we can simply run our playbook in the normal manner. For example, to run it on the development inventory, execute the following:

```
$ ansible-playbook -i inventories/development/hosts site.yml

PLAY [Play using best practise directory structure] ****************************

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

TASK [installapp : Display http_port variable contents] ************************
ok: [app01.dev.example.com] => {
    "http_port": 8080
}
ok: [app02.dev.example.com] => {
    "http_port": 8080
}

TASK [installapp : Create /tmp/foo] ********************************************
changed: [app02.dev.example.com]
changed: [app01.dev.example.com]

TASK [installapp : Use custom module to copy /tmp/foo] *************************
changed: [app02.dev.example.com]
changed: [app01.dev.example.com]

TASK [installapp : Define a fact about automation] *****************************
ok: [app01.dev.example.com]
ok: [app02.dev.example.com]

TASK [installapp : Tell us about automation with a custom filter applied] ******
ok: [app01.dev.example.com] => {
    "msg": "Ansible is an excellent automation tool"
}
ok: [app02.dev.example.com] => {
    "msg": "Ansible is an excellent automation tool"
}

PLAY RECAP *********************************************************************
app01.dev.example.com : ok=6 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
app02.dev.example.com : ok=6 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Similarly, run the following for the production inventory:

```
$ ansible-playbook -i inventories/production/hosts site.yml

PLAY [Play using best practise directory structure] ****************************
```

```
TASK [Gathering Facts] *********************************************************
ok: [app02.prod.example.com]
ok: [app01.prod.example.com]

TASK [installapp : Display http_port variable contents] ************************
ok: [app01.prod.example.com] => {
    "http_port": 80
}
ok: [app02.prod.example.com] => {
    "http_port": 80
}

TASK [installapp : Create /tmp/foo] ********************************************
changed: [app01.prod.example.com]
changed: [app02.prod.example.com]

TASK [installapp : Use custom module to copy /tmp/foo] *************************
changed: [app02.prod.example.com]
changed: [app01.prod.example.com]

TASK [installapp : Define a fact about automation] *****************************
ok: [app01.prod.example.com]
ok: [app02.prod.example.com]

TASK [installapp : Tell us about automation with a custom filter applied] ******
ok: [app01.prod.example.com] => {
    "msg": "Ansible is an excellent automation tool"
}
ok: [app02.prod.example.com] => {
    "msg": "Ansible is an excellent automation tool"
}

PLAY RECAP *********************************************************************
app01.prod.example.com : ok=6 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
app02.prod.example.com : ok=6 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Notice how the appropriate hosts and associated variables are picked up for each inventory and how tidy and well organized our directory structure is. This is the ideal way for you to lay out your playbooks and will ensure that they can be scaled up to whatever size you need them to be, without them becoming unwieldy and difficult to manage or troubleshoot. In the next section of this lab, we will explore the best approaches for working with cloud inventories.

The best approach to cloud inventories

In Lab 3, we looked at a simple example of how you can work with a dynamic inventory, and we walked you through a practical example using the Cobbler provisioning system. However, when it comes to working with cloud inventories (which are simply a form of dynamic inventory, but specifically focused on the cloud), they might, at first, seem somewhat confusing and you may find it difficult to get them up and running. If you follow the high-level procedure outlined in this section, this will become an easy and straightforward task.

As this is a practically focused course, we will choose an example to work with. Sadly, we don't have space to provide practical examples for all of the cloud providers, but if you follow the high-level process we will outline for Amazon EC2 and apply it to your desired cloud provider (for example, Microsoft Azure or Google Cloud Platform), you will find that the process of getting up and running is actually quite straightforward.

An important note before we start, however, is that in Ansible versions up to and including 2.8.x, the dynamic inventory scripts are part of the Ansible source code itself and can be obtained from the main Ansible repository that we examined and cloned previously in this course. With the ever-growing and expanding nature of Ansible, it has become necessary, in the version 2.9.x releases (and beyond), to separate the dynamic inventory scripts into a new distribution mechanism called Ansible collections, which will become mainstream in the 2.10 version (not yet released at the time of writing). You can learn more about Ansible collections and what they are at https://www.ansible.com/blog/getting-started-with-ansible-collections.

The way you download and work with dynamic inventory scripts is likely to change radically with the 2.10 release of Ansible, yet sadly, very little has been revealed, at the time of writing, of what this will look like. As a result, we will guide you through the process of downloading your required dynamic inventory provider scripts for the current 2.9 release, and advise you to consult the Ansible documentation when the 2.10 release comes out for the download location of the relevant scripts. Once you have downloaded them, it is my understanding that you will be able to continue working with them as outlined in this lab.

If you are working with the 2.9 release of Ansible, you can find and download all of the latest dynamic inventory scripts from the stable-2.9 branch on GitHub, at https://github.com/ansible/ansible/tree/stable-2.9/contrib/inventory.

Although the official Ansible documentation has been updated, most guides on the internet still reference the old GitHub locations of these scripts and you will find that they no longer work. Do bear this in mind when working with dynamic inventories! Let's now proceed to cover at the process for working with a dynamic inventory script for a cloud provider; we will use the following Amazon EC2 dynamic inventory script as a working example, but the principles we apply here can equally be used with any other cloud inventory scripts:

1. Having established that we are going to work with Amazon EC2, our first task is to obtain the dynamic inventory script and its associated configuration file. As cloud technologies move fast, it is probably safest to download the latest version of these files directly from the official Ansible project on GitHub. The following three commands will download the dynamic inventory script and make it executable, as well as downloading the template configuration file:

```
$ wget https://raw.githubusercontent.com/ansible/ansible/stable-
2.9/contrib/inventory/ec2.py
$ chmod +x ec2.py
$ wget https://raw.githubusercontent.com/ansible/ansible/stable-
2.9/contrib/inventory/ec2.ini
```

2. With the files successfully downloaded, let's take a look inside them. Unfortunately, Ansible dynamic inventories do not have the same neat documentation system that we have seen in modules and plugins. Fortunately for us, however, the authors of these dynamic inventory scripts have put lots of helpful comments at the top of these files to get us started. Let's take a look inside [ec2.py]:

```
#!/usr/bin/env python

'''
EC2 external inventory script
=================================

Generates inventory that Ansible can understand by making API request to
AWS EC2 using the Boto library.
```

```
NOTE: This script assumes Ansible is being executed where the environment
variables needed for Boto have already been set:
    export AWS_ACCESS_KEY_ID='AK123'
    export AWS_SECRET_ACCESS_KEY='abc123'

Optional region environment variable if region is 'auto'

This script also assumes that there is an ec2.ini file alongside it. To specify
 a
different path to ec2.ini, define the EC2_INI_PATH environment variable:

    export EC2_INI_PATH=/path/to/my_ec2.ini
```

There are pages of documentation to read, but some of the most pertinent information is contained within those first few lines. First of all, we need to ensure that the [Boto] library is installed. Secondly, we need to set the AWS access parameters for [Boto]. The author of this document has given us the quickest way to get started (indeed, it is not their job to replicate the [Boto] documentation).

However, if you refer to the official documentation for [Boto], you'll see that there are lots of ways of configuring it with your AWS credentials---setting the environment variables is just one. You can read more about configuring the [Boto] authentication at https://boto3.amazonaws.com/v1/documentation/api/latest/guide/configuration.html.

3. Before we go ahead and install [Boto], let's take a look at the sample [ec2.ini] file:

```
# Ansible EC2 external inventory script settings
#

[ec2]

# to talk to a private eucalyptus instance uncomment these lines
# and edit edit eucalyptus_host to be the host name of your cloud controller
#eucalyptus = True
#eucalyptus_host = clc.cloud.domain.org

# AWS regions to make calls to. Set this to 'all' to make request to all regions
# in AWS and merge the results together. Alternatively, set this to a comma
# separated list of regions. E.g. 'us-east-1,us-west-1,us-west-2' and do not
# provide the 'regions_exclude' option. If this is set to 'auto', AWS_REGION or
# AWS_DEFAULT_REGION environment variable will be read to determine the region.
regions = all
regions_exclude = us-gov-west-1, cn-north-1
```

Again, you can see pages of well-documented options in this file, and if you scroll all the way to the bottom, you'll even see that you can specify your credentials in this file as an alternative to the methods discussed previously. The default settings for this file are, however, sufficient if you just want to get started.

4. Let's now make sure the [Boto] library is installed; exactly how you do this will depend on your chosen OS and your version of Python. You might be able to install it through a package; on CentOS 7, you can do this as follows:

```
$ sudo apt -y install python-boto python-boto3
```

Alternatively, you can use [pip] for this purpose. For example, to install it as part of your Python 3 environment, you can run the following command:

```
$ sudo pip3 install boto3
```

5. Once you have [Boto] installed, let's go ahead and set our AWS credentials using the environment variables suggested to us in the preceding documentation:

```
$ export AWS_ACCESS_KEY_ID='<YOUR_DATA>'
$ export AWS_SECRET_ACCESS_KEY='<YOUR_DATA>'
```

6. With these steps complete, you can now use your dynamic inventory script in the usual way---you simply reference the executable inventory script with the [-i] parameter in the same way you do with static inventories. For example, if you want to run the Ansible [ping] module as an ad hoc command against all the hosts you have running in Amazon EC2, you would need to run the following command. Make sure you substitute the user account specified by the [-u] switch with the one you connect to your EC2 instances with. Also, reference your private SSH key file:

```
$ ansible -i ec2.py -u ec2-user --private-key /home/james/my-ec2-id_rsa -m ping all
```

That's all there is to it---if you approach all dynamic inventory scripts in this same methodical manner, you will have no problem getting them up and running. Just remember that the documentation is normally embedded in both the script file and its accompanying configuration file, and make sure you read both before you attempt to use the scripts.

One thing to note is that many of the dynamic inventory scripts, [ec2.py] included, cache the results of their API calls to the cloud provider to speed up repeated runs and avoid excessive API calls. However, you might find that in a fast-moving development environment, changes to your cloud infrastructure are not picked up fast enough. For most scripts, there are two ways around this---most feature cache configuration parameters in their configuration file, such as the [cache_path] and [cache_max_age] parameters in [ec2.ini]. If you don't want to set these for every single run, you can also refresh the cache manually by calling the dynamic inventory script directly with a special switch---for example, in [ec2.py]:

```
$ ./ec2.py --refresh-cache
```

That concludes our practical introduction to cloud inventory scripts. As we discussed, provided you consult the documentation (both on the internet and embedded within each dynamic inventory script) and follow the simple methodology we described, you should have no problems and should be able to get up and running with dynamic inventories in minutes. In the next section, we'll revert back to looking at static inventories and the best ways to differentiate your various technology environments.

Differentiating between different environment types

In almost every business, you will need to split your technology environment by type. For example, you will almost certainly have a development environment, where all the testing and development work is performed, and a production environment, where all of the stable test code is run. The environments should (in a best-case scenario) make use of the same Ansible playbooks---after all, the logic is that if you can successfully deploy and test an application in your development environment, then you should be able to deploy it in the same way in a production environment and have it work just as well. However, there are always differences between the two environments, not just in the hostnames, but also sometimes in the parameters, the load balancer names, the port numbers, and so on---the list can seem endless.

In the *The preferred directory layout* section of this lab, we covered a way of differentiating between a development and production environment using two separate inventory directory trees. This is how you should proceed when it comes to differentiating these environments; so, obviously, we won't repeat the examples, but it's important to note that when working with multiple environments, your goals should be as follows:

- Try and reuse the same playbooks for all of your environments that run the same code. For example, if you deploy a web app in your development environment, you should be confident that your playbooks will deploy the same app in the production environment (and your **Quality Assurance** (**QA**) environment, as well as any others that it might need to be deployed in).
- This means that not only are you testing your application deployments and code, you are also testing your Ansible playbooks and roles as part of your overall testing process.
- Your inventories for each environment should be kept in separate directory trees (as we saw in the *The preferred directory layout *section of this lab), but all roles, playbooks, plugins, and modules (if used) should be in the same directory structure (this should be the case for both environments).
- It is normal for different environments to require different authentication credentials; you should keep these separate not only for security but also to ensure that playbooks are not accidentally run in the wrong environment.
- Your playbooks should be in your version control system, just as your code is. This enables you to track changes over time and ensure that everyone is working from the same copy of the automation code.

If you pay attention to these simple pointers, you will find that your automation workflow becomes a real asset to your business and ensures reliability and consistency across all of your deployments. Conversely, failure to follow these pointers puts you at risk of experiencing the dreaded, *it worked in development but it doesn't work in production* deployment failures that so often plague the technology industry. Let's now build on this discussion in the next section by looking at best practices when handling host and group variables, something that, as we saw in *The preferred directory layout* section, you need to apply, especially when working with multiple environments.

The proper approach to defining group and host variables

When working with group and host variables, you can split them up using the directory-based approach we used in the *The preferred directory layout* section. However, there are a few additional pointers to managing this that you should be aware of. First and foremost, you should always pay attention to variable precedence. A detailed list of variable precedence order can be found at https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable. However, the key takeaways for working with multiple environments are as follows:

- Host variables are always of a higher order of precedence than group variables; so, you can override any group variable with a host variable. This behavior is useful if you take advantage of it in a controlled manner, but can yield unexpected results if you are not aware of it.
- There is a special group variables definition called [all], which is applied to all inventory groups. This has a lower order of precedence than specifically defined group variables.
- What happens if you define the same variable twice in two groups? If this happens, both groups have the same order of precedence, so which one wins? To demonstrate this (and our earlier examples), we will create a simple practical example for you to follow.

To get started, let's create a directory structure for our inventories. To keep this example as concise as possible, we will only create a development environment. However, you are free to expand on these concepts by building on the more complete example we covered in the *The preferred directory layout* section of this lab:

1. Create an inventory directory structure with the following commands:

```
$ mkdir -p inventories/development/group_vars
$ mkdir -p inventories/development/host_vars
```

2. Create a simple inventory file with two hosts in a single group in the [inventories/development/hosts] file; the contents should be as follows:

```
[app]
app01.dev.example.com
app02.dev.example.com
```

3. Now, let's create a special group variable file for all the groups in the inventory; this file will be called [inventories/development/group_vars/all.yml] and should contain the following content:

```
---
http_port: 8080
```

4. Finally, let's create a simple playbook called [site.yml] to query and print the value of the variable we just created:

```
---
- name: Play using best practise directory structure
  hosts: all

  tasks:
    - name: Display the value of our inventory variable
      debug:
        var: http_port
```

5. Now, if we run this playbook, we'll see that the variable (which we only defined in one place) takes the value we would expect:

```
$ ansible-playbook -i inventories/development/hosts site.yml

PLAY [Play using best practise directory structure] ****************************

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

TASK [Display the value of our inventory variable] *****************************
ok: [app01.dev.example.com] => {
    "http_port": 8080
}
ok: [app02.dev.example.com] => {
    "http_port": 8080
}

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

6. So far, so good! Now, let's add a new file to our inventory directory structure, with the [all.yml] file remaining unchanged. Let's also create a new file located in [inventories/development/group_vars/app.yml], which will contain the following content:

```
---
http_port: 8081
```

7. We have now defined the same variable twice---once in a special group called [all] and once in the [app] group (which both servers in our development inventory belong to). So, what happens if we now run our playbook? The output should appear as follows:

```
$ ansible-playbook -i inventories/development/hosts site.yml

PLAY [Play using best practise directory structure] ****************************

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

TASK [Display the value of our inventory variable] *****************************
ok: [app01.dev.example.com] => {
    "http_port": 8081
}
ok: [app02.dev.example.com] => {
    "http_port": 8081
}

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

8. As expected, the variable definition in the specific group won, which is in line with the order of precedence documented for Ansible. Now, let's see what happens if we define the same variable twice in two specifically named groups. To complete this example, we'll create a child group, called [centos], and another group that could notionally contain hosts built to a new build standard, called [newcentos], which both application servers will be a member of. This means modifying [inventories/development/hosts] so that it now looks as follows:

```
[app]
app01.dev.example.com
app02.dev.example.com

[centos:children]
app

[newcentos:children]
app
```

9. Now, let's redefine the [http_port] variable for the [centos] group by creating a file called [inventories/development/group_vars/centos.yml], which contains the following content:

```
---
http_port: 8082
```

10. Just to add to the confusion, let's also define this variable for the [newcentos] group in [inventories/development/group_vars/newcentos.yml], which will contain the following content:

```
---
http_port: 8083
```

11. We've now defined the same variable four times at the group level! Let's rerun our playbook and see which value comes through:

```
$ ansible-playbook -i inventories/development/hosts site.yml

PLAY [Play using best practise directory structure] ****************************

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

TASK [Display the value of our inventory variable] *****************************
ok: [app01.dev.example.com] => {
    "http_port": 8083
}
ok: [app02.dev.example.com] => {
    "http_port": 8083
}

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

The value we entered in [newcentos.yml] won---but why? The Ansible documentation states that where identical variables are defined at the group level in the inventory (the one place you can do this), the one from the last-loaded group wins. Groups are processed in alphabetical order and [newcentos] is the group with the name beginning furthest down the alphabet---so, its value of [http_port] was the value that won.

12. Just for completeness, we can override all of this by leaving the [group_vars] directory untouched, but adding a file called [inventories/development/host_vars/app01.dev.example.com.yml], which will contain the following content:

```
---
http_port: 9090
```

13. Now, if we run our playbook one final time, we will see that the value we defined at the host level completely overrides any value that we set at the group level for [app01.dev.example.com]. [app02.dev.example.com] is unaffected as we did not define a host variable for it, so the next highest level of precedence---the group variable from the [newcentos] group---won:

```
$ ansible-playbook -i inventories/development/hosts site.yml

PLAY [Play using best practise directory structure] ****************************

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

TASK [Display the value of our inventory variable] *****************************
ok: [app01.dev.example.com] => {
    "http_port": 9090
}
ok: [app02.dev.example.com] => {
    "http_port": 8083
}

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

With this knowledge, you can now make advanced decisions about how to structure your variables within your inventory to make sure you achieve the desired results at both a host and group level. It's important to know about variable precedence ordering, as these examples have demonstrated, but following the documented order will also allow you to produce powerful, flexible playbook inventories that work well across multiple environments. Now, you may have noticed that, throughout this lab, we have used a top-level playbook in our directory structure called [site.yml]. We will look at this playbook in greater detail in the next section.

Using top-level playbooks

In all of the examples so far, we have built out using the best practice directory structure recommended by Ansible and continually referred to a top-level playbook, typically called [site.yml]. The idea behind this playbook, and, indeed, its common name across all of our directory structures, is so that it can be used across your entire server estate---that is to say, your **site**.

This, of course, is not to say that you have to use the same set of playbooks across every server in your infrastructure or for every single function; rather, it means only you can make the best decision as to what suits your environment best. However, the whole aim of Ansible automation is that the created solution is simple to run and operate. Imagine handing a playbook directory structure with 100 different playbooks to a new system administrator---how would they know which ones to run and in which circumstances? The task of training someone to use the playbooks would be immense and would simply move complexity from one area to another.

At the other the end of the spectrum, you could make use of the [when] clauses with facts and inventory grouping, such that your playbook knows exactly what to run on each server in every possible circumstance. This, of course, is unlikely to happen and the truth is that your automation solution will end up somewhere in the middle.

The most important thing is that, on receipt of a new playbook directory structure, a new operator at least knows what the starting point for both running the playbooks, and understanding the code is. If the top-level playbook they encounter is always [site.yml], then at least everyone knows where to start. Through the clever use of roles and the [import_*] and [include_*] statements, you can split your playbook up into logical portions of reusable code, as we previously discussed, all from one playbook file.

Now that you have learned about the importance of top-level playbooks, let's take a look, in the next section, at how to take advantage of version control tools to ensure good practices are adhered to when it comes to centralizing and maintaining your automation code.

Leveraging version control tools

As we discussed earlier in this lab, it is vital that you version control and test not only your code but also your Ansible automation code. This should include inventories (or dynamic inventory scripts), any custom modules, plugins, roles, and playbook code. The reason for this is simple---the ultimate goal of Ansible automation is likely to be to deploy an entire environment using a playbook (or set of playbooks). This might even involve deploying infrastructure as code, especially if you are deploying to a cloud environment.

Any changes to your Ansible code could mean big changes to your environment, and possibly even whether an important production service works or not. As a result, it is vital that you maintain a version history of your Ansible code and that everyone works from the same version. You are free to choose the version control system that suits you best; most corporate environments will already have some kind of version control system in place. However, if you haven't worked with version control systems before, we recommend that you sign up for a free account on somewhere such as GitHub or GitLab, which both offer version control repositories for free, along with more advanced paid-for plans.

A complete discussion of version control with Git is beyond the scope of this course; there are, indeed, entire books devoted to the subject. However, we will take you through the simplest possible use case. It is assumed, in the following examples, that you are using a free account on GitHub, but if you are using a different provider, simply change the URLs to match those given to you by your version control repository host.

In addition to this, you will need to install the command-line Git tools on your Linux host. On CentOS, you would install these as follows:

```
$ sudo apt install git
```

On Ubuntu, the process is similarly straightforward:

```
$ sudo apt-get update
$ sudo apt-get install git
```

Once the tools are installed and your account is set up, your next task is to clone a Git repository to your machine. If you want to start working with your own repository, you will need to set this up with your provider---excellent documentation is provided by both GitHub and GitLab and you should follow this to set up your first repository.

Once it is set up and initialized, you can clone a copy to your local machine to make changes to your code. This local copy is called a working copy, and you can work through the process of cloning it and making changes as follows (note that these are purely hypothetical examples to give you an idea of the commands you will need to run; you should adapt them for your own use case):

1. Clone your [git] repository to your local machine to create a working copy using a command such as the following:

```
$ git clone https://github.com/<YOUR_GIT_ACCOUNT>/<GIT_REPO>.git
Cloning into '<GIT_REPO>'...
remote: Enumerating objects: 7, done.
remote: Total 7 (delta 0), reused 0 (delta 0), pack-reused 7
Unpacking objects: 100% (7/7), done.
```

2. Change to the directory of the code you cloned (the working copy) and make any code changes you need to make:

```
$ cd <GIT_REPO>
$ vim myplaybook.yml
```

3. Be sure to test your code and, when you are happy with it, add the changed files that are ready for committing a new version using a command such as the following:

```
$ git add myplaybook.yml
```

4. The next step is to commit the changes you have made. A commit is basically a new version of code within the repository, so it should be accompanied by a meaningful [commit] message (specified in quotes after the [-m] switch), as follows:

```
$ git commit -m 'Added new spongle-widget deployment to myplaybook.yml'
[master ed14138] Added new spongle-widget deployment to myplaybook.yml
 Committer: Fenago <ansible@fenago.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 1 insertion(+), 1 deletion(-)
```

5. Right now, all of these changes live solely in the working copy on your local machine. This is good by itself, but it would be better if the code was available to everyone who needs to view it on the version control system. To push your updated commits back to (for example) GitHub, run the following command:

```
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 297 bytes | 297.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/<YOUR_GIT_ACCOUNT>/<GIT_REPO>.git
   0d00263..ed14138 master -> master
```

That's all there is to it!

6. Now, other collaborators can clone your code just as we did in *step 1*. Alternatively, if they already have a working copy of your repository, they can update their working copy using the following command (you can also do this if you want to update your working copy to see changes made by someone else):

```
$ git pull
```

There are some incredibly advanced topics and use cases for Git that are beyond the scope of this course. However, you will find that roughly 80% of the time, the preceding commands are all the Git command-line knowledge you need. There are also a number of graphical frontends to Git, as well as code editors and **Integrated Development Environments** (**IDEs**), that integrate with Git repositories and can assist you further in taking advantage of them. With that complete, let's take a look at how to ensure you can use the same playbook (or role) across multiple hosts, even though they might have different OSes and versions.

Setting OS and distribution variances

Assume that we are using the following simple inventory file for this example, which has two hosts in a single group called [app]:

```
[app]
app01.dev.example.com
app02.dev.example.com
```

Let's now build a simple playbook that demonstrates how you can group differing plays using an Ansible fact so that the OS distribution determines which play in a playbook gets run. Follow these steps to create this playbook and observe it's operation:

1. Start by creating a new playbook---we'll call it [osvariants.yml]---with the following [Play] definition. It will also contain a single task, as shown:

```
---
- name: Play to demonstrate group_by module
  hosts: all

  tasks:
    - name: Create inventory groups based on host facts
      group_by:
        key: os_{{ ansible_facts['distribution'] }}
```

The playbook structure will be, by now, incredibly familiar to you. However, the use of the [group_by] module is new. It dynamically creates new inventory groups based on the key that we specify---in this example, we are creating groups based on a key comprised of the [os_] fixed string, followed by the OS distribution fact obtained from the [Gathering Facts] stage. The original inventory group structure is preserved and unmodified, but all the hosts are also added to the newly created groups according to their facts.

So, the two servers in our simple inventory remain in the [app] group, but if they are based on Ubuntu, they will be added to a newly created inventory group called [os_Ubuntu]. Similarly, if they are based on CentOS, they will be added to a group called [os_CentOS].

2. Armed with this information, we can go ahead and create additional plays based on the newly created groups. Let's add the following [Play] definition to the same playbook file to install Apache on CentOS:

```
- name: Play to install Apache on CentOS
  hosts: os_CentOS
  become: true

  tasks:
    - name: Install Apache on CentOS
      apt:
        name: apache2
        state: present
```

This is a perfectly normal [Play] definition that uses the [apt] module to install the [apache2] package (as required on CentOS). The only thing that differentiates it from our earlier work is the [hosts] definition at the top of the play. This uses the newly created inventory group created by the [group_by] module in the first play.

3. We can, similarly, add a third [Play] definition, this time for installing the [apache2] package on Ubuntu using the [apt] module:

```
- name: Play to install Apache on Ubuntu
  hosts: os_Ubuntu
  become: true

  tasks:
    - name: Install Apache on Ubuntu
      apt:
        name: apache2
        state: present
```

4. If our environment is based on CentOS servers and we run this playbook, the results are as follows:

```
$ ansible-playbook -i hosts osvariants.yml

PLAY [Play to demonstrate group_by module] *************************************

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

TASK [Create inventory groups based on host facts] *****************************
ok: [app01.dev.example.com]
ok: [app02.dev.example.com]

PLAY [Play to install Apache on CentOS] ****************************************

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

TASK [Install Apache on CentOS] ************************************************
changed: [app02.dev.example.com]
changed: [app01.dev.example.com]
[WARNING]: Could not match supplied host pattern, ignoring: os_Ubuntu

PLAY [Play to install Apache on Ubuntu] ****************************************
skipping: no hosts matched

PLAY RECAP *********************************************************************
app01.dev.example.com : ok=4 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

```
app02.dev.example.com : ok=4 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Notice how the task to install Apache on CentOS was run. It was run this way because the [group_by] module created a group called [os_CentOS] and our second play only runs on hosts in the group called [os_CentOS]. As there were no servers running on Ubuntu in the inventory, the [os_Ubuntu] group was never created and so the third play does not run. We receive a warning about the fact that there is no host pattern that matches [os_Ubuntu], but the playbook does not fail---it simply skips this play.

We provided this example to show you another way of managing the inevitable variance in OS types that you will come across in your automation coding. At the end of the day, it is up to you to choose the coding style most appropriate to you. You can make use of the [group_by] module, as detailed here, or write your tasks in blocks and add a [when] clause to the blocks so that they only run when a certain fact-based condition is met (for example, the OS distribution is CentOS)---or perhaps even a combination of the two. The choice is ultimately yours and these different examples are provided to empower you with multiple options that you can choose between to create the best possible solution for your scenario.

Finally, let's round off this lab with a look at porting your automation code between Ansible versions.

Summary

In this lab, you learned about the best practices for directory layout that you should adopt for your playbooks and the steps you should adopt when working with cloud inventories. You then learned new ways of differentiating environments by OS type, as well as more about variable precedence and how to leverage it when working with host and group variables. You then explored the importance of the top-level playbook, before looking at how to make use of version control tools to manage your automation code. Finally, you explored the new techniques for creating single playbooks that will manage servers of different OS versions and distributions, before finally looking at the important topic of porting your code to new Ansible versions.

In the next lab, we will look at some of the more advanced ways that you can use Ansible to take care of some special cases that may arise on your automation journey.

Questions

1. What is a safe and easy way to manage (that is, modify, fix, and create) code changes continuously and share them with others?

A) Playbook revision

B) Task history

C) Ad hoc creation

D) With a Git repository

E) Log management

2. True or false -- Ansible Galaxy supports sharing roles with other users from a central, community-supported repository.

A) True

B) False

3. True or false -- Ansible modules are guaranteed to be available in all future releases of Ansible.

A) True

B) False

Further reading

Manage multiple repositories, versions, or tasks by creating branches and tags to control multiple versions effectively. Refer to the following links for more details:

- How to use Git tagging: https://git-scm.com/course/en/v2/Git-Basics-Tagging
- How to use Git branches: https://git-scm.com/docs/git-branch