Lab 7. Coding Best Practices

In this lab, we will cover the following topics:

- The preferred directory layout
- 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

Lab Environment

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

```
cd ~/Desktop/ansible-course/Lab_7
```

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:

Note: Complete solution is avaialble at: `cd ~/Desktop/ansible-course/Lab_7/best-practise-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: touch
  remote_user: root

- 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 }}"
```

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:

```
$ cd ~/Desktop/ansible-course/Lab_7/best-practise-directory-structure
$ 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.

The proper approach to defining group and host variables

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:

Note: Complete solution is avaialble at: `cd ~/Desktop/ansible-course/Lab_7/variable-precedence-1`

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:

```
$ cd ~/Desktop/ansible-course/Lab_7/variable-precedence-1
$ 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:

```
$ cd ~/Desktop/ansible-course/Lab_7/variable-precedence-2
$ 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:

```
$ cd ~/Desktop/ansible-course/Lab_7/variable-precedence-3
$ 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:

```
$ cd ~/Desktop/ansible-course/Lab_7/variable-precedence-4
$ 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.

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.

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: httpd
        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: [app01.dev.example.com]
ok: [app02.dev.example.com]

TASK [Create inventory groups based on host facts]
*****************************************************************
ok: [app01.dev.example.com]
ok: [app02.dev.example.com]
[WARNING]: Could not match supplied host pattern, ignoring: os_CentOS

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

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

TASK [Gathering Facts]
*****************************************************

ok: [app02.dev.example.com]
ok: [app01.dev.example.com]

TASK [Install Apache on Ubuntu]
*****************************************************
ok: [app02.dev.example.com]
ok: [app01.dev.example.com]

PLAY RECAP
*********************************************************************************************

app01.dev.example.com      : ok=4    changed=0    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0
app02.dev.example.com      : ok=4    changed=0    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, you might get apt-get lock error which can be ignored. This error won't occur using multiple machines

Notice how the task to install Apache on Ubuntu was run. It was run this way because the [group_by] module created a group called [os_Ubuntu] and our second play only runs on hosts in the group called [os_Ubuntu]. As there were no servers running on Ubuntu in the inventory, the [os_CentOS] group was never created and so the third play does not run.

Leveraging version control tools (Optional)

You will need to install the command-line Git tools on your Linux host. On Ubuntu, the process is similarly straightforward:

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

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. Alternatively, if they already have a working copy of your repository, they can update their working copy using the following command:

```
$ git pull
```

Summary

In this lab, you learned about the best practices for directory layout that you should adopt for your playbooks. 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. 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