Lab 4. Playbooks and Roles

Specifically, in this lab, we will cover the following topics:

- Understanding the playbook framework
- Understanding roles---the playbook organizer
- Using conditions in your code
- Repeating tasks with loops
- Grouping tasks using blocks
- Configuring play execution via strategies
- Using [ansible-pull]

Understanding the playbook framework

Although YAML format is easy to read and write, it is very pedantic when it comes to spacing. For example, you cannot use tabs to set indentation even though on the screen a tab and four spaces might look identical---in YAML, they are not. We recommend that you adopt an editor with YAML support to aid you in writing your playbooks if you are doing this for the first time, perhaps Vim, Visual Studio Code, or Eclipse, as these will help you to ensure that your indentation is correct. To test the playbooks we develop in this lab, we will reuse a variant of an inventory created in Lab 3 (unless stated otherwise):

```
[frontends]
frt01.example.com https_port=8443
frt02.example.com http_proxy=proxy.example.com

[frontends:vars]
ntp_server=ntp.frt.example.com
proxy=proxy.frt.example.com

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

[webapp:children]
frontends
apps

[webapp:vars]
proxy_server=proxy.webapp.example.com
health_check_retry=3
health_check_interal=60
```

Let's dive right in and get started writing a playbook.

1. Create a simple playbook to run on the hosts in the [frontends] host group defined in our inventory file. We can set the user that will access the hosts using the [remote_user] directive in the playbook as demonstrated in the following (you can also use the [--user] switch on the command line, but as this lab is about playbook development, we'll ignore that for now):

```
---
- hosts: frontends
```

```
  remote_user: root

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

2. Add another task below the first to run the [shell] module (that will, in turn, run the [ls] command on the remote hosts). We'll also add the [ignore_errors] directive to this task to ensure that our playbook doesn't fail if the [ls] command fails (for example, if the directory we're trying to list doesn't exist). Be careful with the indentation and ensure it matches that of the first part of the file:

```
  - name: run a simple command
    shell: /bin/ls -al /nonexistent
    ignore_errors: True
```

Let's see how our newly created playbook behaves when we run it:

```
$ ansible-playbook -i hosts myplaybook.yaml

PLAY [frontends] ***************************************************************

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

TASK [simple connection test] **************************************************
ok: [frt01.example.com]
ok: [frt02.example.com]

TASK [run a simple command] ****************************************************
fatal: [frt02.example.com]: FAILED! => {"changed": true, "cmd": "/bin/ls -al
/nonexistent", "delta": "0:00:00.015687", "end": "2020-04-10 16:37:56.895520", "msg":
"non-zero return code", "rc": 2, "start": "2020-04-10 16:37:56.879833", "stderr":
"/bin/ls: cannot access /nonexistent: No such file or directory", "stderr_lines":
["/bin/ls: cannot access /nonexistent: No such file or directory"], "stdout": "",
"stdout_lines": []}
...ignoring
fatal: [frt01.example.com]: FAILED! => {"changed": true, "cmd": "/bin/ls -al
/nonexistent", "delta": "0:00:00.012160", "end": "2020-04-10 16:37:56.930058", "msg":
"non-zero return code", "rc": 2, "start": "2020-04-10 16:37:56.917898", "stderr":
"/bin/ls: cannot access /nonexistent: No such file or directory", "stderr_lines":
["/bin/ls: cannot access /nonexistent: No such file or directory"], "stdout": "",
"stdout_lines": []}
...ignoring

PLAY RECAP *********************************************************************
frt01.example.com : ok=3 changed=1 unreachable=0 failed=0 skipped=0 rescued=0
ignored=1
frt02.example.com : ok=3 changed=1 unreachable=0 failed=0 skipped=0 rescued=0
ignored=1
```

From the output of the playbook run, you can see that our two tasks were executed in the order in which they were specified. We can see that the [ls] command failed because we tried to list a directory that did not exist, but the playbook did not register any [failed] tasks because we set [ignore_errors] to [true] for this task (and only this task).

Every module returns a set of results and among these results is the task status. You can see these summarized at the bottom of the preceding playbook run output, and their meaning is as follows:

- [ok]: The task ran successfully and no changes were made.
- [changed]: The task ran successfully and a change was made.
- [failed]: The task failed to run.
- [unreachable]: The host was unreachable to run the task on.
- [skipped]: This task was skipped.
- [ignored]: This task was ignored (for example, in the case of [ignore_errors]).
- [rescued]: We will see an example of this later when we look at blocks and rescue tasks.

These statuses can be very useful---for example, if we have a task to deploy a new Apache configuration file from a template, we know we must restart the Apache service for the changes to be picked up. However, we only want to do this if the file was actually changed---if no changes were made, we don't want to needlessly restart Apache as it would interrupt people who might be using the service. Hence, we can use the [notify] action, which tells Ansible to call a [handler] when (and only when) the result from a task is [changed]. In brief, a handler is a special type of task that is run as a result of a [notify]. However, unlike Ansible playbook tasks, which are performed in sequence, handlers are all grouped together and run at the very end of the play. Also, they can be notified more than once but will only be run once regardless, again preventing needless service restarts. Consider the following playbook:

```
---
- name: Handler demo 1
  hosts: frt01.example.com
  gather_facts: no
  become: yes

  tasks:
    - name: Update Apache configuration
      template:
        src: template.j2
        dest: /etc/apache2/apache2.conf
      notify: Restart Apache

  handlers:
    - name: Restart Apache
      service:
        name: apache2
        state: restarted
```

To keep the output concise, I've turned off fact-gathering for this playbook (we won't use them in any of the tasks). I'm also running this on just one host again for conciseness, but you are welcome to expand the demo code as you wish. If we run this task a first time, we will see the following results:

```
$ ansible-playbook -i hosts handlers1.yml

PLAY [Handler demo 1] ************************************************************

TASK [Update Apache configuration] ***********************************************
changed: [frt01.example.com]
```

```
RUNNING HANDLER [Restart Apache] ******************************************
changed: [frt01.example.com]

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

Notice how the handler was run at the end, as the configuration file was updated. However, if we run this playbook a second time without making any changes to the template or configuration file, we will see something like this:

```
$ ansible-playbook -i hosts handlers1.yml

PLAY [Handler demo 1] *********************************************************

TASK [Update Apache configuration] ********************************************
ok: [frt01.example.com]

PLAY RECAP ********************************************************************
frt01.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

This time, the handler was not called as the result from the configuration task as OK. All handlers should have a globally unique name so that the notify action can call the correct handler. You could also call multiple handlers by setting a common name for using the [listen] directive---this way, you can call either the handler [name] or the [listen] string---as demonstrated in the following example:

```
---
- name: Handler demo 1
  hosts: frt01.example.com
  gather_facts: no
  become: yes

  handlers:
    - name: restart chronyd
      service:
        name: chronyd
        state: restarted
      listen: "restart all services"
    - name: restart apache
      service:
        name: apache2
        state: restarted
      listen: "restart all services"

  tasks:
    - name: restart all services
      command: echo "this task will restart all services"
      notify: "restart all services"
```

We only have one task in the playbook, but when we run it, both handlers are called. Also, remember that we said earlier that [command] was among a set of modules that were a special case because they can't detect whether a change has occurred---as a result, they always return the [changed] value, and so, in this demo playbook, the handlers will always be notified:

```
$ ansible-playbook -i hosts handlers2.yml

PLAY [Handler demo 1] ***********************************************************

TASK [restart all services] ******************************************************
changed: [frt01.example.com]

RUNNING HANDLER [restart chronyd] ************************************************
changed: [frt01.example.com]

RUNNING HANDLER [restart apache] *************************************************
changed: [frt01.example.com]

PLAY RECAP ***********************************************************************
frt01.example.com : ok=3 changed=3 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

These are some of the fundamentals that you need to know to start writing your own playbooks. With these under your belt, let's run through a comparison of ad hoc commands and playbooks in the next section.

Comparing playbooks and ad hoc tasks

Ad hoc commands allow you to quickly create and execute one-off commands, without keeping any record of what was done (other than perhaps your shell history). These serve an important purpose and can be very valuable in getting small changes made quickly and for learning Ansible and its modules.

Playbooks, by contrast, are logically organized sets of tasks (each could conceivably be an ad hoc command), put together in a sequence that performs one bigger action. The addition of conditional logic, error handling, and so on means that, very often, the benefits of playbooks outweigh the usefulness of ad hoc commands. In addition, provided you keep them organized, you will have copies of all previous playbooks that you run and so you will be able to refer back (if ever you need to) to see what you ran and when.

Let's develop a practical example---suppose you want to install Apache 2.4 on CentOS. There are a number of steps involved even if the default configuration is sufficient (which is unlikely, but for now, we'll keep the example simple). If you were to perform the basic installation by hand, you would need to install the package, open up the firewall, and ensure the service is running (and runs at boot time).

To perform these commands in the shell, you might do the following:

```
$ sudo apt install apache2
$ sudo service apache2 start
$ sudo service apache2 status
```

Note: Start will because port 80 is alredy in use by lab environment.

Now, for each of these commands, there is an equivalent ad hoc Ansible command that you could run. We won't go through all of them here in the interests of space; however, let's say you want to restart the Apache service---in this case, you could run an ad hoc command similar to the following (again, we will perform it only on one host for conciseness):

```
$ ansible -i hosts frt01* -m service -a "name=apache2 state=restarted"
```

When run successfully, you will see pages of shell output containing all of the variable data returned from running the service module in this way. A snippet of this is shown in the following for you to check yours against---the key

thing being that the command resulted in the [changed] status, meaning that it ran successfully and that the service was indeed restarted:

```
frt01.example.com | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": true,
    "name": "apache2",
    "state": "started",
```

You could create and execute a series of ad hoc commands to replicate the six shell commands given in the preceding and run them all individually. With a bit of cleverness, you should reduce this from six commands (for example, the Ansible [service] module can both enable a service at boot time and restart it in one ad hoc command). However, you would still ultimately end up with at least three or four ad hoc commands, and if you want to run these again later on another server, you will need to refer to your notes to figure out how you did it.

A playbook is hence a far more valuable way to approach this---not only will it perform all of the steps in one go, but it will also give you a record of how it was done for you to refer to later on. There are multiple ways to do this, but consider the following as an example:

```
---
- name: Install Apache
  hosts: frt01.example.com
  gather_facts: no
  become: yes

  tasks:
    - name: Install Apache package
      apt:
        name: apache2
        state: latest
    - name: Open firewall for Apache
      firewalld:
        service: "{{ item }}"
        permanent: yes
        state: enabled
        immediate: yes
      loop:
        - "http"
        - "https"
    - name: Restart and enable the service
      service:
        name: apache2
        state: restarted
        enabled: yes
```

Now, when you run this, you should see that all of our installation requirements have been completed by one fairly simple and easy to read playbook. There is a new concept here, loops, which we haven't covered yet, but don't worry, we will cover this later in this lab:

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

```
PLAY [Install Apache] ***********************************************************

TASK [Install Apache package] ***************************************************
changed: [frt01.example.com]

TASK [Open firewall for Apache] *************************************************
changed: [frt01.example.com] => (item=http)
changed: [frt01.example.com] => (item=https)

TASK [Restart and enable the service] *******************************************
changed: [frt01.example.com]

PLAY RECAP **********************************************************************
frt01.example.com : ok=2 changed=3 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

As you can see, this is far better for capturing what was actually done and documenting it in a format that someone else could easily pick up. Even though we will cover loops later on in the course, it's fairly easy to see from the preceding how they might be working. With this set out, let's proceed in the next section to look in more detail at a couple of terms we have used several times to ensure you are clear on their meanings: **plays** and **tasks**.

Defining plays and tasks

So far when we have worked with playbooks, we have been creating one single play per playbook (which logically is the minimum you can do). However, you can have more than one play in a playbook, and a "play" in Ansible terms is simply a set of tasks (and roles, handlers, and other Ansible facets) associated with a host (or group of hosts). A task is the smallest possible element of a play and is responsible for running a single module with a set of arguments to achieve a specific goal. Of course, in theory, this sounds quite complex, but when backed up by a practical example, it becomes quite simple to understand.

If we refer to our example inventory, this describes a simple two-tier architecture (we've left out the database tier for now). Now, suppose we want to write a single playbook to configure both the frontend servers and the application servers. We could use two separate playbooks to configure the front end and application servers, but this risks fragmenting your code and making it difficult to organize. However, front end servers and application servers are going to be (by their very nature) fundamentally different and so are unlikely to be configured with the same set of tasks.

The solution to this problem is to create a single playbook with two plays in it. The start of each play can be identified by the line at the lowest indentation (that is, zero spaces in front of it). Let's get started with building up our playbook:

1. Add the first play to the playbook and define some simple tasks to set up the Apache server on the front end, as shown here:

```
---
- name: Play 1 - configure the frontend servers
  hosts: frontends
  become: yes

  tasks:
  - name: Install the Apache package
    apt:
      name: apache2
```

```
      state: latest
  - name: Start the Apache server
    service:
      name: apache2
      state: started
```

2. Immediately below this, in the same file, add the second play to configure the application tier servers:

```
- name: Play 2 - configure the application servers
  hosts: apps
  become: true

  tasks:
  - name: Install Tomcat
    apt:
      name: tomcat
      state: latest
  - name: Start the Tomcat server
    service:
      name: tomcat
      state: started
```

Now, you have two plays: one to install web servers in the [frontends] group and one to install application servers in the [apps] group, all combined into one simple playbook.

When we run this playbook, we'll see the two plays performed sequentially, in the order they appear in the playbook. Note the presence of the [PLAY] keyword, which denotes the start of each play:

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

PLAY [Play 1 - configure the frontend servers] *********************************

TASK [Gathering Facts] *********************************************************
changed: [frt02.example.com]
changed: [frt01.example.com]

TASK [Install the Apache package] **********************************************
changed: [frt01.example.com]
changed: [frt02.example.com]

TASK [Start the Apache server] *************************************************
changed: [frt01.example.com]
changed: [frt02.example.com]

PLAY [Play 2 - configure the application servers] *******************************

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

TASK [Install Tomcat] **********************************************************
changed: [app02.example.com]
changed: [app01.example.com]
```

```
TASK [Start the Tomcat server] ************************************************
changed: [app02.example.com]
changed: [app01.example.com]

PLAY RECAP ********************************************************************
app01.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
app02.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt01.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt02.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

There we have it---one playbook, yet two distinct plays operating on different sets of hosts from the provided inventories. This is very powerful, especially when combined with roles (which will be covered later in this course). Of course, you can have just one play in your playbook---you don't have to have multiple ones, but it is important to be able to develop multi-play playbooks as you will almost certainly find them useful as your environment gets more complex.

Playbooks are the lifeblood of Ansible automation---they extend it beyond single task/commands (which in themselves are incredibly powerful) to a whole series of tasks organized in a logical fashion. As you extend your library of playbooks, however, how do you keep your work organized? How do you efficiently reuse the same blocks of code? In the preceding example, we installed Apache, and this might be a requirement on a number of your servers. However, should you attempt to manage them all from one playbook? Or should you perhaps keep copying and pasting the same block of code over and over again? There is a better way, and in Ansible terms, we need to start looking at roles, which we shall do in the very next section.

Understanding roles -- the playbook organizer

Roles are designed to enable you to efficiently and effectively reuse Ansible code. They always follow a known structure and often will include sensible default values for variables, error handling, handlers, and so on. Taking our Apache installation example from the previous lab, we know that this is something that we might want to do over and over again, perhaps with a different configuration file each time, and perhaps with a few other tweaks on a per-server (or per inventory group) basis. In Ansible, the most efficient way to support the reuse of this code in this way would be to create it as a role.

The process of creating roles is in fact very simple---Ansible will (by default) look within the same directory as you are running your playbook from for a [roles/] directory, and in here, you will create one subdirectory for each role. The role name is derived from the subdirectory name---there is no need to create complex metadata or anything else---it really is that simple. Within each subdirectory goes a fixed directory structure that tells Ansible what the tasks, default variables, handlers, and so on are for each role.

The [roles/] directory is not the only play Ansible will look for roles---this is the first directory it will look in, but it will then look in [/etc/ansible/roles] for any additional roles. This can be further customized through the Ansible configuration file.

Let's explore this in a little more detail. Consider the following directory structure:

```
site.yml
frontends.yml
dbservers.yml
```

```
roles/
   installapache/
     tasks/
     handlers/
     templates/
     vars/
     defaults/
   installtomcat/
     tasks/
     meta/
```

The preceding directory structure shows two roles defined in our hypothetical playbook directory, called [installapache] and [installtomcat]. Within each of these directories, you will notice a series of subdirectories. These subdirectories do not need to exist (more on what they mean in a minute, but for example, if your role has no handlers, then [handlers/] does not need to be created). However, where you do require such a directory, you should populate it with a YAML file named [main.yml]. Each of these [main.yml] files will be expected to have certain contents, depending on the directory that contained them.

The subdirectories that can exist inside of a role are as follows:

- [tasks]: This is the most common directory to find in a role, and it contains all of the Ansible tasks that the role should perform.
- [handlers]: All handlers used in the role should go into this directory.
- [defaults]: All default variables for the role go in here.
- [vars]: These are other role variables---these override those declared in the [defaults/] directory as they are higher up the precedence order.
- [files]: Files needed by the role should go in here---for example, any configuration files that need to be deployed to the target hosts.
- [templates]: Distinct from the [files/] directory, this directory should contain all templates used by the role.
- [meta]: Any metadata needed for the role goes in here. For example, roles are normally executed in the order they are called from the parent playbook---however, sometimes a role will have dependency roles that need to be run first, and if this is the case, they can be declared within this directory.

For the examples we will develop in this part of this lab, we will need an inventory, so let's reuse the inventory we used in the previous section (included in the following for convenience):

```
[frontends]
frt01.example.com https_port=8443
frt02.example.com http_proxy=proxy.example.com

[frontends:vars]
ntp_server=ntp.frt.example.com
proxy=proxy.frt.example.com

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

[webapp:children]
frontends
apps

[webapp:vars]
```

```
proxy_server=proxy.webapp.example.com
health_check_retry=3
health_check_interal=60
```

Let's get started with some practical exercises to help you to learn how to create and work with roles. We'll start by creating a role called [installapache], which will handle the Apache installation process we looked at in the previous section. However, here, we will expand it to cover the installation of Apache on both CentOS and Ubuntu. This is good practice, especially if you are looking to submit your roles back to the community as the more general purpose they are (and the wider the range of systems they will work on), the more useful they will be to people. Step through the following process to create your first role:\

1. Create the directory structure for the [installapache] role from within your chosen playbook directory---this is as simple as this:

```
$ mkdir -p roles/installapache/tasks
```

2. Now, let's create the mandatory [main.yml] inside the [tasks] directory we just created. This won't actually perform the Apache installation---rather, it will call one of two external tasks files, depending on the operating system detected on the target host during the fact-gathering stage. We can use this special variable, [ansible_distribution], in a [when] condition to determine which of the tasks files to import:

```
---
- name: import a tasks based on OS platform
  import_tasks: centos.yml
  when: ansible_distribution == 'CentOS'
- import_tasks: ubuntu.yml
  when: ansible_distribution == 'Ubuntu'
```

2. Create [centos.yml] in [roles/installapache/tasks] to install the latest version of the Apache web server via the [apt] package manager. This should contain the following content:

```
---
- name: Install Apache using apt
  apt:
    name: "apache2"
    state: latest
- name: Start the Apache server
  service:
    name: apache2
    state: started
```

3. Create a file called [ubuntu.yml] in [roles/installapache/tasks] to install the latest version of the Apache web server via the [apt] package manager on Ubuntu. Notice how the content differs between CentOS and Ubuntu hosts:

```
---
- name: Install Apache using apt
  apt:
    name: "apache2"
    state: latest
- name: Start the Apache server
  service:
```

```
    name: apache2
    state: started
```

For now, we're keeping our role code really simple---however, you can see that the preceding tasks files are just like an Ansible playbook, except that they lack the play definition. As they do not come under a play, they are also at a lower indentation level than in a playbook, but apart from this difference, the code should look very familiar to you. In fact, this is part of the beauty of roles: as long as you pay attention to getting the indentation level right, you can more or less use the same code in a playbook or a role.

Now, roles don't run by themselves---we have to create a playbook to call them, so let's write a simple playbook to call our newly created role. This has a play definition just like we saw before, but then rather than having a [tasks:] section within the play, we have a [roles:] section where the roles are declared instead. Convention dictates that this file be called [site.yml], but you are free to call it whatever you like:

```
---
- name: Install Apache using a role
  hosts: frontends
  become: true

  roles:
    - installapache
```

For clarity, your final directory structure should look like this:

```
.
├── roles
│   └── installapache
│   └── tasks
│   ├── centos.yml
│   ├── main.yml
│   └── ubuntu.yml
└── site.yml
```

With this completed, you can now run your [site.yml] playbook using [ansible-playbook] in the normal way---you should see output similar to this:

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

PLAY [Install Apache using a role] *********************************************

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

TASK [installapache : Install Apache using apt] ********************************
changed: [frt02.example.com]
changed: [frt01.example.com]

TASK [installapache : Start the Apache server] *********************************
changed: [frt01.example.com]
changed: [frt02.example.com]

TASK [installapache : Install Apache using apt] ********************************
```

```
skipping: [frt01.example.com]
skipping: [frt02.example.com]

TASK [installapache : Start the Apache server] ********************************
skipping: [frt01.example.com]
skipping: [frt02.example.com]

PLAY RECAP ********************************************************************
frt01.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=2 rescued=0
ignored=0
frt02.example.com : ok=3 changed=2 unreachable=0 failed=0 skipped=2 rescued=0
ignored=0
```

That's it---you have created, at the simplest possible level, your first role. Of course (as we discussed earlier), there is much more to a role than just simple tasks as we have added here, and we will see expanded examples as we work through this lab. However, the preceding example is intended to show you how quick and easy it is to get started with roles.

Before we look at some of the other aspects relating to roles, let's take a look at some other ways to call your role. Ansible allows you to statically import or dynamically include roles when you write a playbook. The syntax between these importing or including a role is subtly different, and notably, both go in the tasks section of your playbook rather than in the roles section. The following is a hypothetical example that shows both options in a really simple playbook. The roles directory structure including both the [common] and [approle] roles would have been created in a similar manner as in the preceding example:

```
---
- name: Play to import and include a role
  hosts: frontends

  tasks:
  - import_role:
      name: common
  - include_role:
      name: approle
```

Setting up role-based variables and dependencies

Variables are at the heart of making Ansible playbooks and roles reusable, as they allow the same code to be repurposed with slightly different values or configuration data. The Ansible role directory structure allows for role-specific variables to be declared in two locations. Although, at first, the difference between these two locations may not seem obvious, it is of fundamental importance.

Roles based variables can go in one of two locations:

- [defaults/main.yml]
- [vars/main.yml]

In addition to the role-based variables described previously, there is also the option to add metadata to a role using the [meta/] directory. As before, to make use of this, simply add a file called [main.yml] into this directory. To explain how you might make use of the [meta/] directory, let's build and run a practical example that will show how it can be used. Before we get started though, it is important to note that, by default, the Ansible parser will only allow you to run a role once. This is somewhat similar to the way in which we discussed handlers earlier, which can be called multiple times but ultimately are only run once at the end of the play. Roles are the same in that they can be or referred to multiple times but will only actually get run once. There are two exceptions to this---the first is if the role

is called more than once but with different variables or parameters, and the other is if the role being called has [allow_duplicates] set to [true] in its [meta/] directory. We shall see examples of both of these as we build our example:

1. At the top level of our practical example, we will have a copy of the same inventory we have been using throughout this lab. We will also create a simple playbook called [site.yml], which contains the following code:

```
---
- name: Role variables and meta playbook
  hosts: frt01.example.com

  roles:
    - platform
```

Notice that we are simply calling one role called [platform] from this playbook---nothing else is called from the playbook itself.

2. Let's go ahead and create the [platform] role---unlike our previous role, this will not contain any tasks or even any variable data; instead, it will just contain a [meta] directory:

```
$ mkdir -p roles/platform/meta
```

Inside this directory, create a file called [main.yml] with the following contents:

```
---
dependencies:
- role: linuxtype
  vars:
    type: centos
- role: linuxtype
  vars:
    type: ubuntu
```

This code will tell Ansible that the platform role is dependent on the [linuxtype] role. Notice that we are specifying the dependency twice, but each type we specify it, we are passing it a variable called [type] with a different value---in this way, the Ansible parser allows us to call the role twice because a different variable value has been passed to it each time it is referred to as a dependency.

3. Let's now go ahead and create the [linuxtype] role---again, this will contain no tasks, but more dependency declarations:

```
$ mkdir -p roles/linuxtype/meta/
```

Again, create a [main.yml] file in the [meta] directory, but this time containing the following:

```
---
dependencies:
- role: version
- role: network
```

Once again, we are creating more dependencies---this time, when the [linuxtype] role is called, it, in turn, is declaring dependencies on roles called [version] and [network].

4. Let's create the [version] role first---this will have both [meta] and [tasks] directories in it:

```
$ mkdir -p roles/version/meta
$ mkdir -p roles/version/tasks
```

In the [meta] directory, we'll create a [main.yml] file with the following contents:

```
---
allow_duplicates: true
```

This declaration is important in this example---as discussed earlier, normally Ansible will only allow a role to be executed once, even if it is called multiple times. Setting [allow_duplicates] to [true] tells Ansible to allow the execution of the role more than once. This is required because, in the [platform] role, we call (via a dependency) the [linuxtype] role twice, which means, in turn, we will call the [version] role twice.

We'll also create a simple [main.yml] file in the tasks directory, which prints the value of the [type] variable that gets passed to the role:

```
---
- name: Print type variable
  debug:
    var: type
```

5. We will now repeat the process with the [network] role---to keep our example code simple, we'll define it with the same contents as the [version] role:

```
$ mkdir -p roles/network/meta
$ mkdir -p roles/network/tasks
```

In the [meta] directory, we'll again create a [main.yml] file with the following contents:

```
---
allow_duplicates: true
```

Again, we'll create a simple [main.yml] file in the [tasks] directory, which prints the value of the [type] variable that gets passed to the role:

```
---
- name: Print type variable
  debug:
    var: type
```

At the end of this process, your directory structure should look like this:

```
.
├── hosts
├── roles
│   ├── linuxtype
│   │   └── meta
│   │       └── main.yml
│   ├── network
│   │   ├── meta
│   │   │   └── main.yml
```

```
│   │   └── tasks
│   │       └── main.yml
│   ├── platform
│   │   └── meta
│   │       └── main.yml
│   └── version
│       ├── meta
│       │   └── main.yml
│       └── tasks
│           └── main.yml
└── site.yml

11 directories, 8 files
```

Let's see what happens when we run this playbook. Now, you might think that the playbook is going to run like this: with the dependency structure we created in the preceding code, our initial playbook statically imports the [platform] role. The [platform] role then states that it depends upon the [linuxtype] role, and the dependency is declared twice with a different value in a variable called [type] each time. The [linuxtype] role then states that it depends upon both the [network] and [version] roles, which are allowed to run more than once and print the value of [type]. Hence, you could be forgiven for thinking that we'll see the [network] and [version] roles called twice, printing [centos] once and [ubuntu] the second time (as this is how we originally specified the dependencies in the [platform] role). However, when we run it, we actually see this:

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

PLAY [Role variables and meta playbook] ************************************

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

TASK [version : Print type variable] ***************************************
ok: [frt01.example.com] => {
    "type": "ubuntu"
}

TASK [network : Print type variable] ***************************************
ok: [frt01.example.com] => {
    "type": "ubuntu"
}

TASK [version : Print type variable] ***************************************
ok: [frt01.example.com] => {
    "type": "ubuntu"
}

TASK [network : Print type variable] ***************************************
ok: [frt01.example.com] => {
    "type": "ubuntu"
}

PLAY RECAP *****************************************************************
```

```
frt01.example.com : ok=5 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

What happened? Although we see that the [network] and [version] roles are called twice (as expected), the value of the [type] variable is always [ubuntu]. This highlights an important point about the way the Ansible parser works and the difference between static imports (which we are doing here) and dynamic includes (which we discussed in the previous section).

With static imports, role variables are scoped as if they were defined at the play level rather than the role level. The roles themselves are all parsed and merged into the play we created in our [site.yml] playbook at parsing time---hence, the Ansible parser creates (in memory) one big playbook that contains all of the merged variable and role content from our directory structure. There is nothing wrong with doing this, but what it means is that the [type] variable gets overwritten each time it is declared, and so the last value we declare (which, in this case, was [ubuntu]) is the value that gets used for the playbook run.

So, how do we get this playbook to run as we originally intended---to load our dependent roles but with the two different values we defined for the [type] variable?

The answer to this question is if we are to continue using statically imported roles, then we should not use role variables when we declare the dependencies. Instead, we should pass over [type] as a role parameter. This is a small but crucial difference---role parameters remain scoped at the role level even when the Ansible parser is run, hence we can declare our dependency twice without the variable getting overwritten. To do this, change the contents of the [roles/platform/meta/main.yml] file to the following:

```
---
dependencies:
- role: linuxtype
  type: centos
- role: linuxtype
  type: ubuntu
```

Do you notice the subtle change? The [vars:] keyword has gone, and the declaration of [type] is now at a lower indentation level, meaning it is a role parameter. Now, when we run the playbook, we get the results that we had hoped for:

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

PLAY [Role variables and meta playbook] ************************************

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

TASK [version : Print type variable] ***************************************
ok: [frt01.example.com] => {
    "type": "centos"
}

TASK [network : Print type variable] ***************************************
ok: [frt01.example.com] => {
    "type": "centos"
}

TASK [version : Print type variable] ***************************************
ok: [frt01.example.com] => {
```

```
    "type": "ubuntu"
}

TASK [network : Print type variable] ***************************************
ok: [frt01.example.com] => {
    "type": "ubuntu"
}

PLAY RECAP *********************************************************************
frt01.example.com : ok=5 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

This is quite an advanced example of Ansible role dependencies but it has been provided to you to demonstrate the importance of knowing a little about variable precedence (that is, where the variable is scoped) and how the parser works. If you write simple, sequentially parsed tasks, then you may never need to know this, but I recommend that you make extensive use of the debug statement and test your playbook design to make sure that you don't fall foul of this during your playbook development.

Having look in great detail at a number of aspects of roles, let's take a look in the following section at a centralized store for publicly available Ansible roles --- Ansible Galaxy.

Ansible Galaxy

No section on Ansible roles would be complete without a mention of Ansible Galaxy. Ansible Galaxy is a community-driven collection of Ansible roles, hosted by Ansible at https://galaxy.ansible.com/. It contains a great many community-contributed Ansible roles, and if you can conceive of an automation task, there is a good chance someone has already written a role to do exactly what you want it to do. It is well worth exploring and can get your automation project off the ground quickly as you can start work with a set of ready-made roles.

In addition to the web site, the [ansible-galaxy] client is included in Ansible, and this provides a quick and convenient way for you to download and deploy roles into your playbook structure. Let's say that you want to update the **message of the day** (**MOTD**) on your target hosts---this is surely something that somebody has already figured out. A quick search on the Ansible Galaxy website returns (at the time of writing) 106 roles for setting the MOTD. If we want to use one of these, we could download it into our roles directory using the following command:

```
$ ansible-galaxy role install -p roles/ arillso.motd
```

That's all you need to do---once the download is complete, you can import or include the role in your playbook just as you would for the manually created roles we have discussed in this lab. Note that if you don't specify [-p roles/], [ansible-galaxy] installs the roles into [~/.ansible/roles], the central roles directory for your user account. This might be what you want, of course, but if you want the role downloaded directly into your playbook directory structure, you would add this parameter.

Another neat trick is to use [ansible-galaxy] to create an empty role directory structure for you to create your own roles in---this saves all of the manual directory and file creation we have been undertaking in this lab, as in this example:

```
$ ansible-galaxy role init --init-path roles/ testrole
- Role testrole was created successfully
$ tree roles/testrole/
roles/testrole/
├── defaults
│   └── main.yml
├── files
```

```
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

Using conditions in your code

In most of our examples so far, we have created simple sets of tasks that always run. However, as you generate tasks (whether in roles or playbooks) that you want to apply to a wider array of hosts, sooner or later, you will want to perform some kind of conditional action. This might be to only perform a task in response to the results of a previous task. Or it might be to only perform a task in response to a specific fact gathered from an Ansible system. In this section, we will provide some practical examples of conditional logic to apply to your Ansible tasks to demonstrate how to use this feature.

As ever, we'll need an inventory to get started, and we'll reuse the inventory we have used throughout this lab:

```
[frontends]
frt01.example.com https_port=8443
frt02.example.com http_proxy=proxy.example.com

[frontends:vars]
ntp_server=ntp.frt.example.com
proxy=proxy.frt.example.com

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

[webapp:children]
frontends
apps

[webapp:vars]
proxy_server=proxy.webapp.example.com
health_check_retry=3
health_check_interal=60
```

Let's define the task that will perform our update but add a [when] clause containing a Jinja 2 expressions to it in a simple example playbook:

```
---
- name: Play to patch only CentOS systems
  hosts: all
```

```
  become: true

  tasks:
  - name: Patch Ubuntu systems
    apt:
      name: apache2
      state: latest
    when: ansible_facts['distribution'] == "Ubuntu"
```

Now, when we run this task, if your test system(s) are Ubuntu-based (and mine are), you should see output similar to the following:

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

PLAY [Play to patch only CentOS systems] ************************************

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

TASK [Patch Ubuntu systems] *************************************************
ok: [app01.example.com]
changed: [frt01.example.com]
ok: [app02.example.com]
ok: [frt02.example.com]

PLAY RECAP ******************************************************************
app01.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
app02.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt01.example.com : ok=2 changed=1 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt02.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

The preceding output shows that all of our systems were Ubuntu-based, but that only [frt01.example.com] needed the patch applying. Now we can make our logic more precise---perhaps it is only our legacy systems that need the patch applying. In this case, we can expand the logic in our playbook to check both the distribution and major version, as follows:

```
---
- name: Play to patch only Ubuntu systems
  hosts: all
  become: true

  tasks:
  - name: Patch Ubuntu systems
    apt:
      name: apache2
      state: latest
```

```
    when: (ansible_facts['distribution'] == "Ubuntu" and
ansible_facts['distribution_major_version'] == "20")
```

Now, if we run our modified playbook, depending on the systems you have in your inventory, you might see output similar to the following. In this case, my [app01.example.com] server was based on CentOS 6 so had the patch applied. All other systems were skipped because they did not match my logical expression:

```
$ ansible-playbook -i hosts condition2.yml

PLAY [Play to patch only Ubuntu systems] ************************************

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

TASK [Patch Ubuntu systems] *************************************************
changed: [app01.example.com]
skipping: [frt01.example.com]
skipping: [frt02.example.com]
skipping: [app02.example.com]

PLAY RECAP ******************************************************************
app01.example.com : ok=2 changed=1 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
app02.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=1 rescued=0
ignored=0
frt01.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=1 rescued=0
ignored=0
frt02.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=1 rescued=0
ignored=0
```

Of course, this conditional logic is not limited to Ansible facts and can be incredibly valuable when using the [shell] or [command] modules. When you run any Ansible module (be it [shell], [command], [apt], [copy], or otherwise), the module returns data detailing the results of its run. You can capture this in a standard Ansible variable using the [register] keyword and then process it further later on in the playbook.

Consider the following playbook code. It contains two tasks, the first of which is to obtain the listing of the current directory and capture the output of the [shell] module in a variable called [shellresult]. When then print a simple [debug] message, but only on the condition that the [hosts] string is in the output of the [shell] command:

```
---
- name: Play to patch only CentOS systems
  hosts: localhost
  become: true

  tasks:
    - name: Gather directory listing from local system
      shell: "ls -l"
      register: shellresult

    - name: Alert if we find a hosts file
```

```
      debug:
        msg: "Found hosts file!"
      when: '"hosts" in shellresult.stdout'
```

Now, when we run this in the current directory, which if you are working from the GitHub repository that accompanies this course will contain a file named [hosts], then you should see output similar to the following:

```
$ ansible-playbook condition3.yml
[WARNING]: provided hosts list is empty, only localhost is available. Note that
the implicit localhost does not match 'all'

PLAY [Play to patch only CentOS systems] ************************************

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

TASK [Gather directory listing from local system] ****************************
changed: [localhost]

TASK [Alert if we find a hosts file] ****************************************
ok: [localhost] => {
    "msg": "Found hosts file!"
}

PLAY RECAP ******************************************************************
localhost : ok=3 changed=1 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

Yet, if the file doesn't exist, then you'll see that the [debug] message gets skipped:

```
$ ansible-playbook condition3.yml
[WARNING]: provided hosts list is empty, only localhost is available. Note that
the implicit localhost does not match 'all'

PLAY [Play to patch only CentOS systems] ************************************

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

TASK [Gather directory listing from local system] ****************************
changed: [localhost]

TASK [Alert if we find a hosts file] ****************************************
skipping: [localhost]

PLAY RECAP ******************************************************************
localhost : ok=2 changed=1 unreachable=0 failed=0 skipped=1 rescued=0 ignored=0
```

You can also create complex conditions for IT operational tasks in production; however, remember that, in Ansible, variables are not cast to any particular type by default, and hence even though the contents of a variable (or fact) might look like a number, Ansible will by default treat it as a string. If you need to perform an integer comparison instead, you must first cast the variable to an integer type. For example, here is a fragment of a playbook that will run a task only on Fedora 25 and newer:

```
tasks:
  - name: Only perform this task on Fedora 25 and later
    shell: echo "only on Fedora 25 and later"
    when: ansible_facts['distribution'] == "Fedora" and
ansible_facts['distribution_major_version']|int >= 25
```

There are many different types of conditionals you can apply to your Ansible tasks, and this section is just scratching the surface; however, it should give you a sound basis on which to expand your knowledge of applying conditions to your tasks in Ansible. Not only can you apply conditional logic to Ansible tasks, but you can also run them in loops over a set of data, and we shall explore this in the next section.

Repeating tasks with loops

Oftentimes, we will want to perform a single task, but use that single task to iterate over a set of data. For example, you might not want to create one user account but 10. Or you might want to install 15 packages to a system. The possibilities are endless, but the point remains the same---you would not want to write 10 individual Ansible tasks to create 10 user accounts. Fortunately, Ansible supports looping over datasets to ensure that you can perform large scale operations using tightly defined code. In this section, we will explore how to make practical use of loops in your Ansible playbooks.

As ever, we must start with an inventory to work against, and we will use our by-now familiar inventory, which we have consistently used throughout this lab:

```
[frontends]
frt01.example.com https_port=8443
frt02.example.com http_proxy=proxy.example.com

[frontends:vars]
ntp_server=ntp.frt.example.com
proxy=proxy.frt.example.com

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

[webapp:children]
frontends
apps

[webapp:vars]
proxy_server=proxy.webapp.example.com
health_check_retry=3
health_check_interal=60
```

Let's start with a really simple playbook to show you how to loop over a set of data in a single task. Although this is quite a contrived example, it is intended to be simple to show you the fundamentals of how loops work in Ansible. We will define a single task that runs the [command] module on a single host from the inventory and uses the [command] module to [echo] the numbers 1 through 6 in turn on the remote system (with some imagination, this could easily be extended to adding user accounts or creating a sequence of files).

Consider the following code:

```
---
- name: Simple loop demo play
  hosts: frt01.example.com

  tasks:
    - name: Echo a value from the loop
      command: echo "{{ item }}"
      loop:
        - 1
        - 2
        - 3
        - 4
        - 5
        - 6
```

The [loop:] statement defines the start of the loop, and the items in the loop are defined as a YAML list. Also, note the higher indentation level, which tells the parser they are part of the loop. When working with the loop data, we use a special variable called [item], which contains the current value from the loop iteration to be echoed. Hence, if we run this playbook, we should see output similar to the following:

```
$ ansible-playbook -i hosts loop1.yml


PLAY [Simple loop demo play] ***************************************************

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

TASK [Echo a value from the loop] **********************************************
changed: [frt01.example.com] => (item=1)
changed: [frt01.example.com] => (item=2)
changed: [frt01.example.com] => (item=3)
changed: [frt01.example.com] => (item=4)
changed: [frt01.example.com] => (item=5)
changed: [frt01.example.com] => (item=6)

PLAY RECAP *********************************************************************
frt01.example.com : ok=2 changed=1 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

You can combine the conditional logic we discussed in the preceding section with loops, to make the loop operate on just a subset of its data. For example, consider the following iteration of the playbook:

```
---
- name: Simple loop demo play
  hosts: frt01.example.com

  tasks:
    - name: Echo a value from the loop
      command: echo "{{ item }}"
      loop:
        - 1
```

```
        - 2
        - 3
        - 4
        - 5
        - 6
      when: item|int > 3
```

Now, when we run this, we see that the task is skipped until we reach the integer value of 4 and higher in the loop contents:

```
$ ansible-playbook -i hosts loop2.yml

PLAY [Simple loop demo play] ***************************************************

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

TASK [Echo a value from the loop] **********************************************
skipping: [frt01.example.com] => (item=1)
skipping: [frt01.example.com] => (item=2)
skipping: [frt01.example.com] => (item=3)
changed: [frt01.example.com] => (item=4)
changed: [frt01.example.com] => (item=5)
changed: [frt01.example.com] => (item=6)

PLAY RECAP *********************************************************************
frt01.example.com : ok=2 changed=1 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

You can, of course, combine this with the conditional logic based on Ansible facts and other variables in the manner we discussed previously. Just as we captured the results of a module's execution using the [register] keyword before, we can do so with loops. The only difference is that the results will now be stored in a dictionary, with one dictionary entry for each iteration of the loop rather than just one set of results.

Hence, let's see what happens if we further enhance the playbook, as follows:

```
---
- name: Simple loop demo play
  hosts: frt01.example.com

  tasks:
    - name: Echo a value from the loop
      command: echo "{{ item }}"
      loop:
        - 1
        - 2
        - 3
        - 4
        - 5
        - 6
      when: item|int > 3
      register: loopresult
```

```
    - name: Print the results from the loop
      debug:
        var: loopresult
```

Now, when we run the playbook, you will see pages out output containing the dictionary with the contents of [loopresult]. The following output is truncated in the interests of space but demonstrates the kind of results you should expect from running this playbook:

```
$ ansible-playbook -i hosts loop3.yml

PLAY [Simple loop demo play] ***************************************************

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

TASK [Echo a value from the loop] **********************************************
skipping: [frt01.example.com] => (item=1)
skipping: [frt01.example.com] => (item=2)
skipping: [frt01.example.com] => (item=3)
changed: [frt01.example.com] => (item=4)
changed: [frt01.example.com] => (item=5)
changed: [frt01.example.com] => (item=6)

TASK [Print the results from the loop] *****************************************
ok: [frt01.example.com] => {
    "loopresult": {
        "changed": true,
        "msg": "All items completed",
        "results": [
            {
                "ansible_loop_var": "item",
                "changed": false,
                "item": 1,
                "skip_reason": "Conditional result was False",
                "skipped": true
            },
            {
                "ansible_loop_var": "item",
                "changed": false,
                "item": 2,
                "skip_reason": "Conditional result was False",
                "skipped": true
            },
```

As you can see, the results section of the output is a dictionary, and we can clearly see that the first two items in the list were [skipped] because the result of our [when] clause ([Conditional]) was [false].

Hence, we can see so far that loops are easy to define and work with---but you may be asking, *can you create nested loops?* The answer to that question is *yes*, but there is a catch---the special variable named [item] would clash as both the inner and outer loops would use the same variable name. This would mean the results from your nested loop run would be, at best, unexpected.

Fortunately, there is a [loop] parameter called [loop_control], which allows you to change the name of the special variable containing the data from the current [loop] iteration from [item] to something of your choosing. Let's create a nested loop to see how this works.

First of all, we'll create a playbook in the usual manner, with a single task to run in a loop. To generate our nested loop, we'll use the [include_tasks] directory to dynamically include a single task from another YAML file that will also contain a loop. As we're intending to use this playbook in a nested loop, we'll use the [loop_var] directive to change the name of the special loop contents variable from [item] to [second_item]:

```
---
- name: Play to demonstrate nested loops
  hosts: localhost

  tasks:
    - name: Outer loop
      include_tasks: loopsubtask.yml
      loop:
        - a
        - b
        - c
      loop_control:
        loop_var: second_item
```

Then, we'll create a second file called [loopsubtask.yml], which contains the inner loop and is included in the preceding playbook. As we're already changed the loop item variable name in the outer loop, we don't need to change it again here. Note that the structure of this file is very much like a tasks file in a role---it is not a complete playbook, but rather simply a list of tasks:

```
---
- name: Inner loop
  debug:
    msg: "second item={{ second_item }} first item={{ item }}"
  loop:
    - 100
    - 200
    - 300
```

Now you should be able to run the playbook, and you will see Ansible iterate over the outer loop first and then process the inner loop over the data defined by the outer loop. As the loop variable names do not clash, all works exactly as we would expect:

```
$ ansible-playbook loopmain.yml
[WARNING]: provided hosts list is empty, only localhost is available. Note that
the implicit localhost does not match 'all'

PLAY [Play to demonstrate nested loops] ************************************

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

TASK [Outer loop] **********************************************************
included: /root/Desktop/ansible-course/Lab_4/loopsubtask.yml for localhost
included: /root/Desktop/ansible-course/Lab_4/loopsubtask.yml for localhost
```

```
included: /root/Desktop/ansible-course/Lab_4/loopsubtask.yml for localhost

TASK [Inner loop] ************************************************************
ok: [localhost] => (item=100) => {
    "msg": "second item=a first item=100"
}
ok: [localhost] => (item=200) => {
    "msg": "second item=a first item=200"
}
ok: [localhost] => (item=300) => {
    "msg": "second item=a first item=300"
}

TASK [Inner loop] ************************************************************
ok: [localhost] => (item=100) => {
    "msg": "second item=b first item=100"
}
ok: [localhost] => (item=200) => {
    "msg": "second item=b first item=200"
}
ok: [localhost] => (item=300) => {
    "msg": "second item=b first item=300"
}

TASK [Inner loop] ************************************************************
ok: [localhost] => (item=100) => {
    "msg": "second item=c first item=100"
}
ok: [localhost] => (item=200) => {
    "msg": "second item=c first item=200"
}
ok: [localhost] => (item=300) => {
    "msg": "second item=c first item=300"
}

PLAY RECAP *******************************************************************
localhost : ok=7 changed=0 unreachable=0 failed=0 skipped=0 rescued=0 ignored=0
```

Loops are simple to work with, and yet very powerful as they allow you to easily use one task to iterate over a large set of data. In the next section, we'll look at another construct of the Ansible language for controlling playbook flow---blocks.

Grouping tasks using blocks

Blocks in Ansible allow you to logically group a set of tasks together, primarily for one of two purposes. One might be to apply conditional logic to an entire set of tasks; in this example, you could apply an identical when clause to each of the tasks, but this is cumbersome and inefficient---far better to place all of the tasks in a block and apply the conditional logic to the block itself. In this way, the logic only needs to be declared once. Blocks are also valuable when it comes to error handling and especially when it comes to recovering from an error condition. We shall explore both of these through simple practical examples in this lab to get you up to speed with blocks in Ansible.

As ever, let's ensure we have an inventory to work from:

```
[frontends]
frt01.example.com https_port=8443
frt02.example.com http_proxy=proxy.example.com

[frontends:vars]
ntp_server=ntp.frt.example.com
proxy=proxy.frt.example.com

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

[webapp:children]
frontends
apps

[webapp:vars]
proxy_server=proxy.webapp.example.com
health_check_retry=3
health_check_interal=60
```

Now, let's dive straight in and look at an example of how you would use blocks to apply conditional logic to a set of tasks. At a high level, suppose we want to perform the following actions on all of our Fedora Linux hosts:

- Install the package for the Apache web server.
- Install a templated configuration.
- Start the appropriate service.

We could achieve this with three individual tasks, all with a [when] clause associated with them, but blocks provide us with a better way. The following example playbook shows the three tasks discussed contained in a block (notice the additional level of indentation required to denote their presence in the block):

```
---
- name: Conditional block play
  hosts: all
  become: true

  tasks:
  - name: Install and configure Apache
    block:
      - name: Install the Apache package
        dnf:
          name: apache2
          state: installed
      - name: Install the templated configuration to a dummy location
        template:
          src: templates/src.j2
          dest: /tmp/my.conf
      - name: Start the apache2 service
        service:
          name: apache2
          state: started
```

```
          enabled: True
    when: ansible_facts['distribution'] == 'Fedora'
```

When you run this playbook, you should find that the Apache-related tasks are only run on any Fedora hosts you might have in your inventory; you should see that either all three tasks are run or are skipped---depending on the makeup and contents of your inventory, it might look something like this:

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

PLAY [Conditional block play] ************************************************

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

TASK [Install the Apache package] ********************************************
changed: [frt01.example.com]
changed: [frt02.example.com]
skipping: [app01.example.com]
skipping: [app02.example.com]

TASK [Install the templated configuration to a dummy location] *****************
changed: [frt01.example.com]
changed: [frt02.example.com]
skipping: [app01.example.com]
skipping: [app02.example.com]

TASK [Start the apache2 service] *********************************************
changed: [frt01.example.com]
changed: [frt02.example.com]
skipping: [app01.example.com]
skipping: [app02.example.com]

PLAY RECAP *******************************************************************
app01.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=3 rescued=0
ignored=0
app02.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=3 rescued=0
ignored=0
frt01.example.com : ok=4 changed=3 unreachable=0 failed=0 skipped=3 rescued=0
ignored=0
frt02.example.com : ok=4 changed=3 unreachable=0 failed=0 skipped=3 rescued=0
ignored=0
```

This is very simple to construct, but very powerful in terms of the effect it has on your ability to control the flow over large sets of tasks.

This time, let's build a different example to demonstrate how blocks can be utilized to help Ansible to handle error conditions gracefully. So far, you should have experienced that if your playbooks encounter any errors, they are likely to stop executing at the point of failure. This is in some situations far from ideal, and you might want to perform some kind of recovery actions in this event rather than simply halting the playbook.

Let's create a new playbook, this time with the following contents:

```
---
- name: Play to demonstrate block error handling
  hosts: frontends

  tasks:
    - name: block to handle errors
      block:
        - name: Perform a successful task
          debug:
            msg: 'Normally executing....'
        - name: Deliberately create an error
          command: /bin/whatever
        - name: This task should not run if the previous one results in an error
          debug:
            msg: 'Never print this message if the above command fails!!!!'
      rescue:
        - name: Catch the error (and perform recovery actions)
          debug:
            msg: 'Caught the error'
        - name: Deliberately create another error
          command: /bin/whatever
        - name: This task should not run if the previous one results in an error
          debug:
            msg: 'Do not print this message if the above command fails!!!!'
      always:
        - name: This task always runs!
          debug:
            msg: "Tasks in this part of the play will be ALWAYS executed!!!!"
```

Notice that in the preceding play, we now have additional sections to [block]---as well as the tasks in [block] itself, we have two new parts labeled [rescue] and [always]. The flow of execution is as follows:

1. All tasks in the [block] section are executed normally, in the sequence in which they are listed.
2. If a task in the [block] results in an error, no further tasks in the [block] are run:
    - Tasks in the [rescue] section start to run in the order they are listed.
    - Tasks in the [rescue] section do not run if no errors result from the [block] tasks.
3. If an error results from a task being run in the [rescue] section, no further [rescue] tasks are executed and execution moves on to the [always] section.
4. Tasks in the [always] section are always run, regardless of any errors in either the [block] or [rescue] sections. They even run when no errors are encountered.

With this flow of execution in mind, you should see output similar to the following when you execute this playbook, noting that we have deliberately created two error conditions to demonstrate the flow:

```
$ ansible-playbook -i hosts blocks-error.yml

PLAY [Play to demonstrate block error handling] ********************************

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

```
TASK [Perform a successful task] ************************************************
ok: [frt01.example.com] => {
    "msg": "Normally executing...."
}
ok: [frt02.example.com] => {
    "msg": "Normally executing...."
}

TASK [Deliberately create an error] *********************************************
fatal: [frt01.example.com]: FAILED! => {"changed": false, "cmd": "/bin/whatever",
"msg": "[Errno 2] No such file or directory", "rc": 2}
fatal: [frt02.example.com]: FAILED! => {"changed": false, "cmd": "/bin/whatever",
"msg": "[Errno 2] No such file or directory", "rc": 2}

TASK [Catch the error (and perform recovery actions)] ***************************
ok: [frt01.example.com] => {
    "msg": "Caught the error"
}
ok: [frt02.example.com] => {
    "msg": "Caught the error"
}

TASK [Deliberately create another error] ****************************************
fatal: [frt01.example.com]: FAILED! => {"changed": false, "cmd": "/bin/whatever",
"msg": "[Errno 2] No such file or directory", "rc": 2}
fatal: [frt02.example.com]: FAILED! => {"changed": false, "cmd": "/bin/whatever",
"msg": "[Errno 2] No such file or directory", "rc": 2}

TASK [This task always runs!] ***************************************************
ok: [frt01.example.com] => {
    "msg": "Tasks in this part of the play will be ALWAYS executed!!!!"
}
ok: [frt02.example.com] => {
    "msg": "Tasks in this part of the play will be ALWAYS executed!!!!"
}

PLAY RECAP **********************************************************************
frt01.example.com : ok=4 changed=0 unreachable=0 failed=1 skipped=0 rescued=1
ignored=0
frt02.example.com : ok=4 changed=0 unreachable=0 failed=1 skipped=0 rescued=1
ignored=0
```

Ansible has two special variables, which contain information you might find useful in the rescue block to perform your recovery actions:

- [ansible_failed_task]: This is a dictionary containing details of the task from [block] that failed, causing us to enter the [rescue] section. You can explore this by displaying its contents using [debug], but for example, the name of the failing task can be obtained from [ansible_failed_task.name].
- [ansible_failed_result]: This is the result of the failed task and behaves the same as if you had added the [register] keyword to the failing task. This saves you having to add [register] to every single task in the block in case it fails.

As your playbooks get more complex and error handling gets more and more important (or indeed conditional logic becomes more vital), blocks will become an important part of your arsenal in writing good, robust playbooks. Let's proceed in the next section to explore execution strategies to gain further control of your playbook runs.

Configuring play execution via strategies

As your playbooks become increasingly complex, it becomes more and more important that you have robust ways to debug any issues that might arise. For example, is there a way you can check the contents of a given variable (or variables) during execution without the need to insert [debug] statements throughout your playbook? Similarly, we have so far seen that Ansible will ensure that a particular task runs to completion on all inventory hosts that it applies to before moving on to the next task---is there a way to vary this?

When you are getting started with Ansible, the execution strategy that you see by default (and we have seen this so far in every playbook we have executed, even though we have not mentioned it by name) is known as [linear]. This does exactly what it describes---each task is executed in turn on all applicable hosts before the next task is started. However, there is another, less commonly used strategy called [free], which allows all tasks to be completed as fast as they can on each host, without waiting for other hosts.

The most useful strategy when you are starting work with Ansible, however, is going to be the [debug] strategy, and this enables Ansible to drop you straight into its integrated debug environment if an error should occur in the playbook. Let's demonstrate this by creating a playbook that has a deliberate error in it. Note the [strategy: debug] and [debugger: on_failed] statements in the play definition:

```
---
- name: Play to demonstrate the debug strategy
  hosts: frt01.example.com
  strategy: debug
  debugger: on_failed
  gather_facts: no
  vars:
    username: daniel

  tasks:
    - name: Generate an error by referencing an undefined variable
      ping: data={{ mobile }}
```

Now if you execute this playbook, you should see that it starts to run, but then drops you into the integrated debugger when it encounters the deliberate error it contains. The start of the output should be similar to the following:

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

PLAY [Play to demonstrate the debug strategy] **********************************

TASK [Generate an error by referencing an undefined variable] ******************
fatal: [frt01.example.com]: FAILED! => {"msg": "The task includes an option with an
undefined variable. The error was: 'mobile' is undefined\n\nThe error appears to be in
'/root/Desktop/ansible-course/Lab_4/debug.yml': line 11, column 7, but may\nbe
elsewhere in the file depending on the exact syntax problem.\n\nThe offending line
appears to be:\n\n tasks:\n - name: Generate an error by referencing an undefined
variable\n ^ here\n"}
[frt01.example.com] TASK: Generate an error by referencing an undefined variable
```

```
(debug)>


[frt02.prod.com] TASK: make an error with refering incorrect variable (debug)> p
task_vars
{'ansible_check_mode': False,
 'ansible_current_hosts': [u'frt02.prod.com'],
 'ansible_diff_mode': False,
 'ansible_facts': {},
 'ansible_failed_hosts': [],
 'ansible_forks': 5,
...
[frt02.prod.com] TASK: make an error with refering incorrect variable (debug)> quit
User interrupted execution
$
```

Notice that the playbook starts executing but fails on the first task with an error as the variable is undefined. However, rather than exiting back to the shell, it enters an interactive debugger. An exhaustive guide to the use of the debugger is beyond the scope of this course, but further details are available here if you are interested in learning: https://docs.ansible.com/ansible/latest/user_guide/playbooks_debugger.html.

To take you through a very simple, practical debugging example, however, enter the [p task] command at the prompt---this will cause the Ansible debugger to print the name of the failing task; this is very useful if you are in the midst of a large playbook:

```
[frt01.example.com] TASK: Generate an error by referencing an undefined variable
(debug)> p task
TASK: Generate an error by referencing an undefined variable
```

Now we know where the play failed, so let's dig a little deeper by issuing the [p task.args] command, which will show us the arguments that were passed to the module in the task:

```
[frt01.example.com] TASK: Generate an error by referencing an undefined variable
(debug)> p task.args
{u'data': u'{{ mobile }}'}
```

So, we can see that our module was passed the argument called [data], with the argument value being a variable (denoted by the pairs of curly braces) called [mobile]. Hence, it might be logical to have a look at the variables available to the task, to see whether this variable exists, and if so whether the value is sensible (use the [p task_vars] command to do this):

```
[frt01.example.com] TASK: Generate an error by referencing an undefined variable
(debug)> p task_vars
{'ansible_check_mode': False,
 'ansible_current_hosts': [u'frt01.example.com'],
 'ansible_dependent_role_names': [],
 'ansible_diff_mode': False,
 'ansible_facts': {},
 'ansible_failed_hosts': [],
 'ansible_forks': 5,
```

The preceding output is truncated, and you will find a great many variables associated with the task---this is because any gathered facts, and internal Ansible variables, are all available to the task. However, if you scroll through the list,

you will be able to confirm that there is no variable called [mobile].

Hence, this should be enough information to fix your playbook. Enter [q] to quit the debugger:

```
[frt01.example.com] TASK: Generate an error by referencing an undefined variable
(debug)> q
User interrupted execution
$
```

The Ansible debugger is an incredibly powerful tool and you should learn to make effective use of it, especially as your playbook complexity grows. This concludes our practical look at the various aspects of playbook design---in the next section, we'll take a look at the ways in which you can integrate Git source code management into your playbooks.

Using ansible-pull

The [ansible-pull] command is a special feature of Ansible that allows you to, all in one go, pull a playbook from a Git repository (for example, GitHub) and then execute it, hence saving the usual steps such as cloning (or updating the working copy of) the repository, then executing the playbook. The great thing about [ansible-pull] is that it allows you to centrally store and version control your playbooks and then execute them with a single command, hence enabling them to be executed using the [cron] scheduler without the need to even install the Ansible playbooks on a given box.

An important thing to note, however, is that, while the [ansible] and [ansible-playbook] commands can both operate over an entire inventory and run the playbooks against one or more remote hosts, the [ansible-pull] command is only intended to run the playbooks it obtains from your source control system on the localhost. Hence, if you want to use [ansible-pull] throughout your infrastructure, you must install it onto every host that needs it.

Nonetheless, let's see how this might work. We'll simply run the command by hand to explore its application, but in reality, you would almost certainly install it into your [crontab] so that it runs on a regular basis, picking up any changes you make to your playbook in the version control system.

As [ansible-pull] is only intended to run the playbook on the local system, an inventory file is somewhat redundant---instead, we'll use a little-used inventory specification whereby you can simply specify inventory hosts directory as a comma-separated list on the command line. If you only have one host, you simply specify its name followed by a comma.

Let's use a simple playbook from GitHub that sets the message of the day based on variable content. To do this, we will run the following command (which we'll break down in a minute):

```
$ ansible-pull -d /var/ansible-set-motd -i ${HOSTNAME}, -U
https://github.com/jamesfreeman959/ansible-set-motd.git site.yml -e
"ag_motd_content='MOTD generated by ansible-pull'" >> /tmp/ansible-pull.log 2>&1
```

This command breaks down as follows:

- [-d /var/ansible-set-motd]: This sets the working directory that will contain the checkout of the code from GitHub.
- [-i ${HOSTNAME},]: This runs only on the current host, specified by its hostname from the appropriate shell variable.
- [-U https://github.com/jamesfreeman959/ansible-set-motd.git]: We use this URL to obtain the playbooks.
- [site.yml]: This is the name of the playbook to run.
- [-e "ag_motd_content='MOTD generated by ansible-pull'"]: This sets the appropriate Ansible variable to generate the MOTD content.

- [>> /tmp/ansible-pull.log 2>&1]: This redirects the output of the command to a log file in case we need to analyze it later---especially useful if running the command in a [cron job] where the output would never be printed to the user's terminal.

When you run this command, you should see some output similar to the following (note that log redirection has been removed to make it easier to see the output):

```
$ ansible-pull -d /var/ansible-set-motd -i ${HOSTNAME}, -U
https://github.com/jamesfreeman959/ansible-set-motd.git site.yml -e
"ag_motd_content='MOTD generated by ansible-pull'"
Starting Ansible Pull at 2020-04-14 17:26:21
/usr/bin/ansible-pull -d /var/ansible-set-motd -i cookbook, -U
https://github.com/jamesfreeman959/ansible-set-motd.git site.yml -e
ag_motd_content='MOTD generated by ansible-pull'
cookbook |[WARNING]: SUCCESS = Your git > {
    "aversion isfter": "7d too old t3a191ecb2do fully suebe7f84f4fpport the
a5817b0f1bdepth argu49c4cd54",ment.
Fall
    "ansing back tible_factso full che": {
     ckouts.
   "discovered_interpreter_python": "/usr/bin/python"
    },
    "before": "7d3a191ecb2debe7f84f4fa5817b0f1b49c4cd54",
    "changed": false,
    "remote_url_changed": false
}

PLAY [Update the MOTD on hosts] ***********************************************

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

TASK [ansible.motd : Add 99-footer file] **************************************
skipping: [cookbook]

TASK [ansible.motd : Delete 99-footer file] ***********************************
ok: [cookbook]

TASK [ansible.motd : Delete /etc/motd file] ***********************************
skipping: [cookbook]

TASK [ansible.motd : Check motd tail supported] *******************************
fatal: [cookbook]: FAILED! => {"changed": true, "cmd": "test -f /etc/update-motd.d/99-
footer", "delta": "0:00:00.004444", "end": "2020-04-14 17:26:25.489793", "msg": "non-
zero return code", "rc": 1, "start": "2020-04-14 17:26:25.485349", "stderr": "",
"stderr_lines": [], "stdout": "", "stdout_lines": []}
...ignoring

TASK [ansible.motd : Add motd tail] *******************************************
skipping: [cookbook]

TASK [ansible.motd : Add motd] ************************************************
changed: [cookbook]
```

```
PLAY RECAP *********************************************************************
cookbook : ok=4 changed=2 unreachable=0 failed=0 skipped=3 rescued=0 ignored=1
```

This command can be a very powerful part of your overall Ansible solution, especially as it means you don't have to worry too greatly about running all of your playbooks centrally, or ensuring that they are all up to date every time you run them. The ability to schedule this in [cron] is especially powerful in a large infrastructure where, ideally, automation means things should take care of themselves.

This concludes our practical look at playbooks and how to author your own code---with a little research into Ansible modules, you should now have enough to write your own robust playbooks with ease.

Summary

In this lab, you learned about the playbook framework and how to start building your own playbooks. You then learned how to organize your code into roles and design your code to effectively and efficiently support reuse. We then explored some of the more advanced playbook writing topics such as working with conditional logic, blocks, and loops. Finally, we looked at playbook execution strategies, especially with a view to being able to debug your playbooks effectively, and we wrapped up with a look at how you can run Ansible playbooks on a local machine directly from GitHub.

In the next lab, we will learn how to consume and create our very own modules, providing you with the skills you need to expand the capabilities of Ansible to suit your own bespoke environments, and to contribute back to the community.

Questions

1. How do you restart the Apache web server in the [frontends] host group via an ad hoc command?

A) [ansible frontends -i hosts -a "name=apache2 state=restarted"]

B) [ansible frontends -i hosts -b service -a "name=apache2 state=restarted"]

C) [ansible frontends -i hosts -b -m service -a "name=apache2 state=restarted"]

D) [ansible frontends -i hosts -b -m server -a "name=apache2 state=restarted"]

E) [ansible frontends -i hosts -m restart -a "name=apache2"]

2. Do blocks allow you to logically make a group of tasks, or perform error handling?

A) True

B) False

3. Default strategies are intended via the relevant modules in the playbook.

A) True

B) False