Lab 8. Advanced Ansible Topics

In this lab, we will cover the following topics:

- Asynchronous versus synchronous actions
- Controlling play execution for rolling updates
- Configuring the maximum failure percentage
- Setting task execution delegation
- Using the [run_once] option
- Running playbooks locally
- Working with proxies and jump hosts
- Configuring playbook prompts
- Placing tags in the plays and tasks
- Securing data with Ansible Vault

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_8
```

Pre-req:

Make sure that following entries exist in `/root/hosts` file before starting the lab:

```
cat /root/hosts
```

```
[frontends]
frt01.example.com
frt02.example.com
```

Asynchronous versus synchronous actions

Ansible tasks can be run asynchronously. Let's explore this through a practical example. Suppose we have two servers in a simple INI-formatted inventory:

```
[frontends]
frt01.example.com
frt02.example.com
```

Now, in order to simulate a long-running task, we'll run the [sleep] command using the [shell] module. However, rather than have it run with the SSH connection blocked for the duration of the [sleep] command, we'll add two special parameters to the task, as shown:

```
---
- name: Play to demonstrate asynchronous tasks
  hosts: frontends
  become: true

  tasks:
    - name: A simulated long running task
      shell: "sleep 20"
```

```
        async: 30
        poll: 5
```

The two new parameters are [async] and [poll]. The [async] parameter tells Ansible that this task should be run asynchronously (so that the SSH connection will not be blocked) for a maximum of [30] seconds.

When you run this playbook, you will find that it runs just like any other playbook; from the terminal output, you won't be able to see any difference. But behind the scenes, Ansible checks the task every [5] seconds until it succeeds or reaches the [async] timeout value of [30] seconds:

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

PLAY [Play to demonstrate asynchronous tasks] *********************************

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

TASK [A simulated long running task] *****************************************
changed: [frt02.example.com]
changed: [frt01.example.com]

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

If you want to check on the task later (that is, if [poll] is set to [0]), you could add a second task to your playbook so that it looks as follows:

```
---
- name: Play to demonstrate asynchronous tasks
  hosts: frontends
  become: true

  tasks:
    - name: A simulated long running task
      shell: "sleep 20"
      async: 30
      poll: 0
      register: long_task

    - name: Check on the asynchronous task
      async_status:
        jid: "{{ long_task.ansible_job_id }}"
      register: async_result
      until: async_result.finished
      retries: 30
```

In this playbook, the initial asynchronous task is defined as before, except we have now set [poll] to [0]. We have also chosen to register the result of this task to a variable called [long_task]---this is so that we can query the job ID for the task when we check it later on. The next (new) task in the play uses the [async_status] module to check on the job

ID we registered from the first task and loops until the job either finishes or reaches [30] retries---whichever comes first. When using these in a playbook, you almost certainly wouldn't add the two tasks back to back like this--- usually, you would perform additional tasks in between them---but to keep this example simple, we will run the two tasks sequentially. Running this playbook should yield an output similar to the following:

```
$ ansible-playbook -i hosts async2.yml

PLAY [Play to demonstrate asynchronous tasks] *********************************

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

TASK [A simulated long running task] *****************************************
changed: [frt02.example.com]
changed: [frt01.example.com]

TASK [Check on the asynchronous task] ****************************************
FAILED - RETRYING: Check on the asynchronous task (30 retries left).
FAILED - RETRYING: Check on the asynchronous task (30 retries left).
FAILED - RETRYING: Check on the asynchronous task (29 retries left).
FAILED - RETRYING: Check on the asynchronous task (29 retries left).
FAILED - RETRYING: Check on the asynchronous task (28 retries left).
FAILED - RETRYING: Check on the asynchronous task (28 retries left).
FAILED - RETRYING: Check on the asynchronous task (27 retries left).
FAILED - RETRYING: Check on the asynchronous task (27 retries left).
changed: [frt01.example.com]
changed: [frt02.example.com]

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

In the preceding code block, we can see that the long-running task is left running and the next task polls its status until the conditions we set are met. In this case, we can see that the task finished successfully and the overall play result was successful.

Control play execution for rolling updates

By default, Ansible parallelizes tasks on multiple hosts. The setting for this is defined by the [forks] parameter in the Ansible configuration file, which defaults to [5] (so, by default, Ansible attempts to run its automation job on five hosts at the same time).

1. Create the following simple playbook to run two commands on the two hosts in our inventory. The content of the command is not important at this stage, but if you run the [date] command using the [command] module, you will be able to see the time that each task is run, as well as if you specify [-v] to increase the verbosity when you run the play:

```
---
- name: Simple serial demonstration play
```

```
  hosts: frontends
  gather_facts: false

  tasks:
    - name: First task
      command: date
    - name: Second task
      command: date
```

2. Now, if you run this play, you will see that it performs all the operations on each host simultaneously, as we have fewer hosts than the default number of forks --- [5]. This behavior is normal for Ansible, but not really what we want as our users will experience service outage:

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

PLAY [Simple serial demonstration play] **************************************

TASK [First task] ************************************************************
changed: [frt02.example.com]
changed: [frt01.example.com]

TASK [Second task] ***********************************************************
changed: [frt01.example.com]
changed: [frt02.example.com]


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

3. Now, let's modify the play definition, as shown. We'll leave the [tasks] sections exactly as they were in *step 1*:

```
---
- name: Simple serial demonstration play
  hosts: frontends
  serial: 1
  gather_facts: false
```

4. Notice the presence of the [serial: 1] line. This tells Ansible to complete the play on [1] host at a time before moving on to the next. If we run the play again, we can see this in action:

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

PLAY [Simple serial demonstration play] **************************************

TASK [First task] ************************************************************
changed: [frt01.example.com]

TASK [Second task] ***********************************************************
changed: [frt01.example.com]
```

```
PLAY [Simple serial demonstration play] ****************************************

TASK [First task] *************************************************************
changed: [frt02.example.com]

TASK [Second task] ************************************************************
changed: [frt02.example.com]

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

You can even build on this by passing a list to the [serial] directive. Consider the following code:

```
  serial:
    - 1
    - 3
    - 5
```

This tells Ansible to run the play on [1] host, initially, then on the next [3], and then on batches of [5] at a time until the inventory is completed.

Configuring the maximum failure percentage

For our practical example, let's consider an expanded inventory with [10] hosts in it. We'll define this as follows:

```
[frontends]
frt[01:10].example.com
```

Now, let's create a simple playbook to run on these hosts. We will set our batch size to [5] and [max_fail_percentage] to [50%] in the play definition:

1. Create the following play definition to demonstrate the use of the [max_fail_percentage] directive:

```
---
- name: A simple play to demonstrate use of max_fail_percentage
  hosts: frontends
  gather_facts: no
  serial: 5
  max_fail_percentage: 50
```

The number of failed hosts must exceed the value of [max_fail_percentage]; if it is equal, the play continues. So, in our example, if exactly 50% of our hosts failed, the play would still continue.

2. Next, we will define two simple tasks. The first task has a special clause under it that we use to deliberately simulate a failure---this line starts with [failed_when] and we use it to tell the task that if it runs this task on the first three hosts in the batch, then it should deliberately fail this task regardless of the result; otherwise, it should allow the task to run as normal:

```
  tasks:
    - name: A task that will sometimes fail
      debug:
        msg: This might fail
      failed_when: inventory_hostname in ansible_play_batch[0:3]
```

3. Finally, we'll add a second task that will always succeed. This is run if the play is allowed to continue, but not if it is aborted:

```
    - name: A task that will succeed
      debug:
        msg: Success!
```

So, we have deliberately constructed a playbook that will run on a 10-host inventory in batches of 5 hosts at a time, but the play is aborted if more than 50% of the hosts in any given batch experiences a failure. We have also deliberately set up a failure condition that causes three of the hosts in the first batch of 5 (60%) to fail.

4. Run the playbook and let's observe what happens:

```
$ ansible-playbook -i morehosts maxfail.yml

PLAY [A simple play to demonstrate use of max_fail_percentage] *****************

TASK [A task that will sometimes fail] ****************************************
fatal: [frt01.example.com]: FAILED! => {
    "msg": "This might fail"
}
fatal: [frt02.example.com]: FAILED! => {
    "msg": "This might fail"
}
fatal: [frt03.example.com]: FAILED! => {
    "msg": "This might fail"
}
ok: [frt04.example.com] => {
    "msg": "This might fail"
}
ok: [frt05.example.com] => {
    "msg": "This might fail"
}

NO MORE HOSTS LEFT ***********************************************************

NO MORE HOSTS LEFT ***********************************************************

PLAY RECAP ******************************************************************
frt01.example.com : ok=0 changed=0 unreachable=0 failed=1 skipped=0 rescued=0
ignored=0
frt02.example.com : ok=0 changed=0 unreachable=0 failed=1 skipped=0 rescued=0
ignored=0
frt03.example.com : ok=0 changed=0 unreachable=0 failed=1 skipped=0 rescued=0
ignored=0
frt04.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
```

```
ignored=0
frt05.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Setting task execution delegation

Although we still want to run our play across our entire inventory, we certainly don't want to run the load balancer commands from those hosts. Let's once again explain this in more detail with a practical example. We'll reuse the two simple host inventories that we used earlier in this lab:

```
[frontends]
frt01.example.com
frt02.example.com
```

Now, to work on this, let's create two simple shell scripts in the same directory as our playbook. These are only examples as setting up a load balancer is beyond the scope of this course. However, imagine that you have a shell script (or other executables) that you can call that can add and remove hosts to and from a load balancer:

1. For our example, let's create a script called [remove_from_loadbalancer.sh], which will contain the following:

```
#!/bin/sh
echo Removing $1 from load balancer...
```

2. We will also create a script called [add_to_loadbalancer.sh], which will contain the following:

```
#!/bin/sh
echo Adding $1 to load balancer...
```

Obviously, in a real-world example, there would be much more code in these scripts!

3. Now, let's create a playbook that will perform the logic we outlined here. We'll first create a very simple play definition (you are free to experiment with the [serial] and [max_fail_percentage] directives as you wish) and an initial task:

```
---
- name: Play to demonstrate task delegation
  hosts: frontends

  tasks:
    - name: Remove host from the load balancer
      command: ./remove_from_loadbalancer.sh {{ inventory_hostname }}
      args:
        chdir: "{{ playbook_dir }}"
      delegate_to: localhost
```

4. After this, we add a task where the upgrade work is carried out. This task has no [delegate_to] directive, and so it is actually run on the remote host from the inventory (as desired):

```
    - name: Deploy code to host
      debug:
        msg: Deployment code would go here....
```

5. Finally, we add the host back to the load balancer using the second script we created earlier. This task is almost identical to the first:

```
- name: Add host back to the load balancer
  command: ./add_to_loadbalancer.sh {{ inventory_hostname }}
  args:
    chdir: "{{ playbook_dir }}"
  delegate_to: localhost
```

6. Let's see this playbook in action:

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

PLAY [Play to demonstrate task delegation] ************************************

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

TASK [Remove host from the load balancer] ************************************
changed: [frt02.example.com -> localhost]
changed: [frt01.example.com -> localhost]

TASK [Deploy code to host] ***************************************************
ok: [frt01.example.com] => {
    "msg": "Deployment code would go here...."
}
ok: [frt02.example.com] => {
    "msg": "Deployment code would go here...."
}

TASK [Add host back to the load balancer] ************************************
changed: [frt01.example.com -> localhost]
changed: [frt02.example.com -> localhost]

PLAY RECAP *******************************************************************
frt01.example.com : ok=4 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt02.example.com : ok=4 changed=2 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Also, note that you can choose to use a form of shorthand notation in your playbooks (and roles) for [delegate_to], called [local_action]. This allows you to specify a task on a single line that would ordinarily be run with [delegate_to: localhost] added below it. Wrapping this all up into a second example, our playbook will look as follows:

```
---
- name: Second task delegation example
  hosts: localhost

  tasks:
  - name: Perform an rsync from localhost to inventory hosts
    local_action: command rsync -a /tmp/ {{ inventory_hostname }}:/tmp/target/
```

The preceding shorthand notation is equivalent to the following:

```
tasks:
  - name: Perform an rsync from localhost to inventory hosts
    command: rsync -a /tmp/ {{ inventory_hostname }}:/tmp/target/
    delegate_to: localhost
```

If we run this playbook, we can see that [local_action] does indeed run [rsync] from [localhost], enabling us to efficiently copy whole directory trees across to remote servers in the inventory:

```
$ ansible-playbook -i localhost delegate2.yml

PLAY [Second task delegation example] *****************************************

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

TASK [Perform an rsync from localhost to inventory hosts] **********************
changed: [frt02.example.com -> localhost]
changed: [frt01.example.com -> localhost]

PLAY RECAP *******************************************************************
frt01.example.com : ok=2 changed=1 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt02.example.com : ok=2 changed=1 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 *file has vanished* error which can be ignored. This error won't occur using multiple machines

Using the run_once option

You can make use of the special [run_once] directive for any tasks you want to run only once on your inventory. For example, let's reuse the 10-host inventory that we defined earlier in this lab. Now, let's proceed to demonstrate this option, as follows:

1. Create the simple playbook as in the following code block. We're using a debug statement to display some output, but in real life, you would insert your script or command that performs your one-off cluster function here (for example, upgrading a database schema):

```
---
- name: Play to demonstrate the run_once directive
  hosts: frontends

  tasks:
    - name: Upgrade database schema
      debug:
        msg: Upgrading database schema...
      run_once: true
```

2. Now, let's run this playbook and see what happens:

```
$ ansible-playbook -i morehosts runonce.yml

PLAY [Play to demonstrate the run_once directive] ******************************

TASK [Gathering Facts] *********************************************************
ok: [frt02.example.com]
ok: [frt05.example.com]
ok: [frt03.example.com]
ok: [frt01.example.com]
ok: [frt04.example.com]
ok: [frt06.example.com]
ok: [frt08.example.com]
ok: [frt09.example.com]
ok: [frt07.example.com]
ok: [frt10.example.com]

TASK [Upgrade database schema] *************************************************
ok: [frt01.example.com] => {
    "msg": "Upgrading database schema..."
}
---

PLAY RECAP *********************************************************************
frt01.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt02.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt03.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt04.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt05.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt06.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt07.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt08.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt09.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt10.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Notice that, just as desired, although the playbook was run on all 10 hosts (and, indeed, gathered facts from all 10 hosts), we only ran the upgrade task on one host.

3. It's important to note that the [run_once] option applies per batch of servers, so if we add [serial: 5] to our play definition (running our play in two batches of 5 on our inventory of 10 servers), the schema upgrade task actually runs twice! It runs once as requested, but once per batch of servers, not once for the entire inventory. Be careful of this nuance when working with this directive in a clustered environment.

Add [serial: 5] to your play definition and rerun the playbook. The output should appear as follows:

```
$ ansible-playbook -i morehosts runonce.yml

PLAY [Play to demonstrate the run_once directive] ******************************

TASK [Gathering Facts] ********************************************************
ok: [frt04.example.com]
ok: [frt01.example.com]
ok: [frt02.example.com]
ok: [frt03.example.com]
ok: [frt05.example.com]

TASK [Upgrade database schema] ************************************************
ok: [frt01.example.com] => {
    "msg": "Upgrading database schema..."
}

PLAY [Play to demonstrate the run_once directive] ******************************

TASK [Gathering Facts] ********************************************************
ok: [frt08.example.com]
ok: [frt06.example.com]
ok: [frt07.example.com]
ok: [frt10.example.com]
ok: [frt09.example.com]

TASK [Upgrade database schema] ************************************************
ok: [frt06.example.com] => {
    "msg": "Upgrading database schema..."
}

PLAY RECAP ********************************************************************
frt01.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt02.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt03.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt04.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt05.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt06.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt07.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt08.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt09.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
frt10.example.com : ok=1 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Running playbooks locally

It is important to note that when we talk about running a playbook locally with Ansible, it is not the same as talking about running it on [localhost]. If we run a playbook on [localhost], Ansible actually sets up an SSH connection to [localhost] (it doesn't differentiate its behavior or attempt to detect whether a host in the inventory is local or remote---it simply tries faithfully to connect).

Indeed, we can try creating a [local] inventory file with the following contents:

```
[local]
localhost
```

Now, if we attempt to run the [ping] module in an ad hoc command against this inventory, we see the following:

Password: **fenago**

```
$ ansible -i localhosts -m ping all --ask-pass


SSH password:
localhost | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

We can now modify our inventory so that it looks as follows:

```
[local]
localhost ansible_connection=local
```

We've added a special variable to our [localhost] entry---the [ansible_connection] variable---which defines which protocol is used to connect to this inventory host. So, we have told it to use a direct local connection instead of an SSH-based connectivity (which is the default).

It should be noted that this special value for the [ansible_connection] variable actually overrides the hostname you have put in your inventory. So, if we change our inventory to look as follows, Ansible will not even attempt to connect to the remote host called [frt01.example.com---]it will connect locally to the machine running the playbook (without SSH):

```
[local]
frt01.example.com ansible_connection=local
```

We can demonstrate this very simply. Let's first check for the absence of a test file in our local [/tmp] directory:

```
ls -l /tmp/foo
ls: cannot access /tmp/foo: No such file or directory
```

Now, let's run an ad hoc command to touch this file on all hosts in the new inventory we just defined:

```
$ ansible -i localhosts2 -m file -a "path=/tmp/foo state=touch" all

frt01.example.com | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": true,
    "dest": "/tmp/foo",
    "gid": 0,
    "group": "root",
    "mode": "0644",
    "owner": "root",
    "size": 0,
    "state": "file",
    "uid": 0
}
```

The command ran successfully, so let's see whether the test file is present on the local machine:

```
$ ls -l /tmp/foo
-rw-r--r-- 1 root root 0 Apr 24 16:28 /tmp/foo
```

It is! So, the ad hoc command did not attempt to connect to [frt01.example.com], even though this host name was in the inventory. The presence of [ansible_connection=local] meant that this command was run on the local machine without using SSH.

Working with proxies and jump hosts

Let's assume that you have two Cumulus Networks switches in your network (these are based on a special distribution of Linux for switching hardware, which is very similar to Debian). These two switches have the [cmls01.example.com] and [cmls02.example.com] hostnames, but both can only be accessed from a host called [bastion.example.com].

The configuration to support our [bastion] host is performed in the inventory, rather than in the playbook. We begin by defining an inventory group with the switches in, in the normal manner:

```
[switches]
cmls01.example.com
cmls02.example.com
```

However, we can now start to get clever by adding some special SSH arguments into the inventory variables for this group. Add the following code to your inventory file:

```
[switches:vars]
ansible_ssh_common_args='-o ProxyCommand="ssh -W %h:%p -q bastion.example.com"'
```

This special variable content tells Ansible to add extra options when it sets up an SSH connection, including to proxy via the [bastion.example.com] host. The [-W %h:%p] options tell SSH to proxy the connection and to connect to the host specified by [%h] (this is either [cmls01.example.com] or [cmls02.example.com]) on the port specified by [%p] (usually port [22]).

Now, if we attempt to run the Ansible [ping] module against this inventory, we can see whether it works:

```
$ ansible -i switches -m ping all

cmls02.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
127.0.0.1 app02.example.com
    "ping": "pong"
}
cmls01.example.com | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python"
    },
    "changed": false,
    "ping": "pong"
}
```

You will notice that we can't actually see any differences in Ansible's behavior from the command-line output. On the surface, Ansible works just as it normally does and connects successfully to the two hosts. However, behind the scenes it proxies via [bastion.example.com].

Configuring playbook prompts

Let's reuse the two host frontend inventories we defined at the beginning of this lab. Now, let's demonstrate how to capture data from users during a playbook run with a practical example:

1. Create a simple play definition in the usual manner, as follows:

```
---
- name: A simple play to demonstrate prompting in a playbook
  hosts: frontends
```

2. Now, we'll add a special section to the play definition. We previously defined a [vars] section, but this time we will define one called [vars_prompt] (which enables you to do just that---define variables through user prompts). In this section, we will prompt for two variables---one for a user ID and one for a password. One will be echoed to the screen, while the other won't be, by setting [private: yes]:

```
vars_prompt:
  - name: loginid
    prompt: "Enter your username"
    private: no
  - name: password
    prompt: "Enter your password"
    private: yes
```

3. We'll now add a single task to our playbook to demonstrate this prompting process of setting the variables:

```
tasks:
  - name: Proceed with login
```

```
      debug:
        msg: "Logging in as {{ loginid }}..."
```

4. Now, let's run the playbook and see how it behaves:

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

Enter your username: root
Enter your password: fenago

PLAY [A simple play to demonstrate prompting in a playbook] ********************

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

TASK [Proceed with login] *****************************************************
ok: [frt01.example.com] => {
    "msg": "Logging in as james..."
}
ok: [frt02.example.com] => {
    "msg": "Logging in as james..."
}

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

Let's proceed to the next section and look at how you can selectively run your tasks from within your plays with the use of tags.

Placing tags in the plays and tasks

In this section we will build a simple playbook with two --- each bearing a different tag to show you how tags work. We will work with the two simple host inventories that we worked with previously:

1. Create the following simple playbook to perform two tasks---one to install the [nginx] package and the other to deploy a configuration file from a template:

```
---
- name: Simple play to demonstrate use of tags
  hosts: frontends

  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present
      tags:
        - install
```

```
    - name: Install nginx configuration from template
      template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx.conf
      tags:
        - customize
```

2. Now, let's run the playbook in the usual manner, but with one difference---this time, we'll add the [--tags] switch to the command line. This switch tells Ansible to only run the tasks that have tags matching the ones that are specified. So, for example, run the following command:

```
$ ansible-playbook -i hosts tags.yml --tags install

PLAY [Simple play to demonstrate use of tags] *********************************

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

TASK [Install nginx] **********************************************************
changed: [frt02.example.com]
changed: [frt01.example.com]

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

Notice that the task to deploy the configuration file doesn't run. This is because it is tagged with [customize] and we did not specify this tag when running the playbook.

3. There is also a [--skip-tags] switch that does the reverse of the previous switch---it tells Ansible to skip the tags listed. So, if we run the playbook again but skip the [customize] tag, we should see an output similar to the following:

```
$ ansible-playbook -i hosts tags.yml --skip-tags customize

PLAY [Simple play to demonstrate use of tags] *********************************

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

TASK [Install nginx] **********************************************************
ok: [frt02.example.com]
ok: [frt01.example.com]

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

```
frt02.example.com : ok=2 changed=0 unreachable=0 failed=0 skipped=0 rescued=0
ignored=0
```

Note that if you don't specify either [--tags] or [--skip-tags], then all the tasks are run, regardless of their tag.

Add [--list-tasks] to your command, which --- rather than running the playbook --- lists the tasks from the playbook that would perform if you run it. Some examples are provided for you in the following code block, based on the example playbook we just created:

```
$ ansible-playbook -i hosts tags.yml --skip-tags customize --list-tasks

playbook: tags.yml

  play #1 (frontends): Simple play to demonstrate use of tags TAGS: []
    tasks:
      Install nginx TAGS: [install]

$ ansible-playbook -i hosts tags.yml --tags install,customize --list-tasks

playbook: tags.yml

  play #1 (frontends): Simple play to demonstrate use of tags TAGS: []
    tasks:
      Install nginx TAGS: [install]
      Install nginx configuration from template TAGS: [customize]

$ ansible-playbook -i hosts tags.yml --list-tasks

playbook: tags.yml

  play #1 (frontends): Simple play to demonstrate use of tags TAGS: []
    tasks:
      Install nginx TAGS: [install]
      Install nginx configuration from template TAGS: [customize]
```

As you can see, not only does [--list-tasks] show you which tasks would run, it also shows you which tags are associated with them, which helps you further understand how tagging works and ensure that you achieve the playbook flow that you wanted.

Securing data with Ansible Vault

Let's proceed with a simple example that shows you how you can use Ansible Vault:

1. Start by creating a new vault to store sensitive data in; we'll call this file [secret.yml]. You can create this using the following command:

Note: Use password : `fenago` . You can also choose any password.

```
$ ansible-vault create secret.yml
New Vault password:
Confirm New Vault password:
```

Enter the password you have chosen for the vault when prompted and confirm it by entering it a second time.

2. When you have entered the password, you will be set to your normal editor (defined by the [EDITOR] shell variable). On my test system, this is [vi]. Within this editor, you should create a [vars] file, in the normal manner, containing your sensitive data:

```
---
secretdata: "Ansible is cool!"
```

3. Save and exit the editor (press *Esc*, then [:wq] in [vi]). You will exit to the shell. Now, if you look at the contents of your file, you will see that they are encrypted and are safe from anyone who shouldn't be able to read the file:

```
$ cat secret.yml

$ANSIBLE_VAULT;1.1;AES256
333437636132616132356163663038333865303833365633839303232653335313238656435326339
326131393566363636663633336613338316238316334610a30306530663634346265393830653 5
643763666343364316264343133653834363832626633383738363263337653930303630333139636 5
656430663136373061 0a336430313536396137636266373735653836656566653666323130616 4
303665393134373166303861356131613033335333061623465326636663938653664
```

4. However, the great thing about Ansible Vault is that you can use this encrypted file in a playbook as if it were a normal [variables] file (although, obviously, you have to tell Ansible your vault password). Let's create a simple playbook as follows:

```
---
- name: A play that makes use of an Ansible Vault
  hosts: frontends

  vars_files:
    - secret.yml

  tasks:
    - name: Tell me a secret
      debug:
        msg: "Your secret data is: {{ secretdata }}"
```

The [vars_files] directive is used in exactly the same way as it would be if you were using an unencrypted [variables] file. Ansible reads the headers of the [variables] files at run time and determines whether they are encrypted or not.

5. Try running the playbook without telling Ansible what the vault password is---in this instance, you should receive an error such as this:

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

ERROR! Attempting to decrypt but no vault secrets found
```

6. Ansible correctly understands that we are trying to load a [variables] file that is encrypted with [ansible-vault], but we must manually tell it the password for it to proceed. There are a number of ways of specifying passwords for vaults (more on this in a minute), but for simplicity, try running the following command and enter your vault password when prompted:

```
$ ansible-playbook -i hosts vaultplaybook.yml --ask-vault-pass
Vault password:

PLAY [A play that makes use of an Ansible Vault] ******************************

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

TASK [Tell me a secret] ******************************************************
ok: [frt01.example.com] => {
    "msg": "Your secret data is: Ansible is cool!"
}
ok: [frt02.example.com] => {
    "msg": "Your secret data is: Ansible is cool!"
}

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

One thing to note is that you don't actually have to have a separate vault file for your sensitive data; you can actually include it inline in your playbook. For example, let's try re-encrypting our sensitive data for inclusion in an otherwise unencrypted playbook. Run the following command in your shell (it should produce an output similar to what is shown):

```
$ ansible-vault encrypt_string 'Ansible is cool!' --name secretdata

New Vault password:
Confirm New Vault password:
secretdata: !vault |
          $ANSIBLE_VAULT;1.1;AES256

3266643630333139623861356462646161613036633537616462363330636138646235356135613166

3732383332663266373962633962396331653632346334650a323136323937623765636566656130

3432356232623566616535303231306231666462353331316362323031396339343636363461356333

3264623737383632350a366563383933330353166643133366335643032383865363135323236436
          303465303935653337613234383933356539386665326131633763343864343303064
Encryption successful
```

You can copy and paste the output of this command into a playbook. So, if we modify our earlier example, it would appear as follows:

```
---
- name: A play that makes use of an Ansible Vault
  hosts: frontends
```

```
  vars:
    secretdata: !vault |
          $ANSIBLE_VAULT;1.1;AES256

3266643630333139623861356462646161303663353761646236333063613864623535613561
3732383332663266373962633962396331653632346334650a3231363239376237656365666656130
3432356232623566616535303231306231666462353333131636232303139636393436363461356333
3264623737383632350a36656338393330353166643133366335643032383865363313532232363436
          3034653039356533376132343839333565393866653261316337633438643430366d
```

  tasks:
    - name: Tell me a secret
      debug:
        msg: "Your secret data is: {{ secretdata }}"
```

Now, when you run this playbook in exactly the same manner as we did before (specifying the vault password using a user prompt), you should see that it runs just as when we used an external encrypted [variables] file:

```
$ ansible-playbook -i hosts inlinevaultplaybook.yml --ask-vault-pass

Vault password:

PLAY [A play that makes use of an Ansible Vault] *******************************

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

TASK [Tell me a secret] *******************************************************
ok: [frt01.example.com] => {
    "msg": "Your secret data is: Ansible is cool!"
}
ok: [frt02.example.com] => {
    "msg": "Your secret data is: Ansible is cool!"
}

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

Summary

In this lab, you learned about running tasks asynchronously in Ansible, before looking at the various features available for running playbooks to upgrade a cluster, such as running tasks on small batches of inventory hosts, failing a play early if a certain percentage of hosts fail, delegating tasks to a specific host, and even running tasks once, regardless of your inventory (or batch) size.

You also learned about the difference between running playbooks locally as opposed to on [localhost] and how to use SSH-proxying to automate tasks on an isolated network via a [bastion] host. Finally, you learned about handling sensitive data without storing it unencrypted at rest, either through prompting the user at run time or through the use of Ansible Vault. You even learned about running a subset of your playbook tasks with tagging.

In the next lab, we will explore a topic we touched on briefly in this lab in more detail---automating network device management with Ansible.

Questions

1. Which parameter allows you to configure the maximum number of hosts in a batch that will fail before a play is aborted?

A) [percentage]

B) [max_fail]

C) [max_fail_percentage]

D) [max_percentage]

E) [fail_percentage]

2. True or false -- you can use the [--connect=local] parameter to run any playbooks locally without using SSH:

A) True

B) False

3. True or false -- in order to run a playbook asynchronously, you need to use the [async] keyword:

A) True

B) False