Guide to working with Ansible

You are now up and running with Ansible in your system, which is otherwise known as the **control host**. The control host is the Ansible host you to dispatch tasks to the remote managed Ansible hosts.

Before you start delegating tasks to a managed host, make sure you have non-root, a sudo-enabled user on that host—it's always a bad idea to connect to a remote VPS via a root user.

Ansible inventory files

The Ansible inventory file lists which hosts will receive commands from the control host. The inventory can list individual hosts, or group them under categories you distinguish.

The default location for the inventory file is `/etc/ansible/hosts`, but it's also possible to change the location of the inventory file by uncommenting the `inventory` parameter in `/etc/ansible/ansible.cfg`

A typical inventory file can list the managed host either by IP address or by domain names. It is also possible to list one managed host in more than one group. Here's an example of listing two hosts under the `webservers` and `dbservers` categories.

```
[webservers]
123.45.67.89
SUBDOMAIN.DOMAIN.TLD

[dbservers]
123.45.67.89
SUBDOMAIN.DOMAIN.TLD
```

To test if all the hosts are discoverable by the inventory file, use the following ad-hoc command.

```
$ ansible all --list-hosts
  hosts (2):
    123.45.67.89
    SUBDOMAIN.DOMAIN.TLD
```

You can also list the hosts by group name:

```
$ ansible dbservers --list-hosts
hosts (2):
    123.45.67.89
    SUBDOMAIN.DOMAIN.TLD
```

Ad-hoc commands in Ansible are merely those that perform a single command across one or many hosts. They don't use tasks but allow you to do a lot of things quite easily without building out playbooks (more on those in the second part of this guide).

To find out if all the hosts are up and running, use the following ad-hoc command that uses the `ping` module of Ansible. The "-u"switch specifies which user Ansible will connect to via SSH—change it according to the non-root user you created earlier.

```
$ ansible all -m ping -u USER

123.45.67.89 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
...
...
```

The `"changed": false` in the above JSON result tells us that the `ping` Ansible task didn't change anything on the remote server.

Rather than specifying all the hosts as in the above command, you can also ping a group of hosts. Specify the group name in place of 'all' with the following command:

```
$ ansible webservers -m ping -u USER
```

Ansible modules

Modules are the discrete units of code that can be used from the terminal or in a playbook task. They simplify Ansible tasks by installing software, copying files, using templates, and so on.

Modules use the available context to determine what actions if any needed to bring the managed host to the desired state and are idempotent, that means if you run the same task again and again, the state of the machine will not change.

To find the list of available modules, use the following command:

```
$ ansible-doc -l
```

Let's try to install Nginx on an Ubuntu/Debian host using an ad-hoc command in Ansible:

```
$ ansible webservers -b --become-user=root -m shell -a 'apt -y install
nginx' -u USER

172.104.160.8 | SUCCESS | rc=0 >>
Reading package lists...
Building dependency tree...
```

The following flags were used with the above command:

- `-b`: Instruct ansible to become another user to run the command
- `--become-user=root`: Run the command as a `root` user
- `-m`: Declares which module is used in the command
- `-a`: Declares which arguments are passed to the module

The alternate and preferred way of installing software using an ad-hoc command is to use `apt` module. If your remote managed host is running RHEL/CentOS, then change the module name from `apt` to `yum`.

```
$ ansible webservers -b --become-user=root -m apt -a 'name=nginx
state=present update_cache=true' -u ansadm
172.104.160.8 | SUCCESS => {
    "cache_update_time": 1530378409,
    "cache_updated": true,
    "changed": true,
    "stderr": "",
    "stderr_lines": [],
...
...
```

In the above Ansible command, the

- a

switch passes the arguments to the `apt` module by specifying the name of the package to be installed, the desired state, and whether to update the package repository cache or not.

The line `change: true` in the result section of the above ad-hoc command signifies that the state of the system has been changed. If you run the above ad-hoc command again, the value of `changed` field will be `false`, which means the state of the system remains unchanged, because Ansible is aware that Nginx is already present in the system and will not try to alter the state again.

That's what we call Ansible **idempotent**. You can run the same ad-hoc command as many times as you'd like and it won't change anything unless it needs to.

```
172.104.160.8 | SUCCESS => {
    "cache_update_time": 1530378676,
    "cache_updated": true,
    "changed": false
}
```

So far, we have understood the ansible modules and its usages through ad-hoc way, but this is not so useful until we use the modules in ansible playbooks to run multiple tasks in the remote managed host.

Tasks in Ansible

When you dispatch a job from a control host to a managed host using one an Ansible module, it is known as a **task**. Tasks can be implemented using ad-hoc commands, as we've done just above, or you can use an Ansible **playbook** (more on those in a moment).

One example of a task is copying a file from the control host to a managed host, since it requires the use of 'copy' module. There are thousands of modules in Ansible, which means a task can use any of the modules to bring a managed host to the desired state. How many modules are there by default in Ansible? Let's see:

```
$ ansible-doc -l | wc -l
1852
```

If you haven't guessed, there are a lot of things you can do when combining Ansible tasks and modules.

Play in Ansible

An Ansible **play** is a set of tasks that are run on one or more managed hosts. A play may include one or many different tasks, and the most common way to execute a play is to use a playbook.

Ansible Playbooks

Ansible **Playbooks** are composed of one or more plays and offer more advanced functionality for sending tasks to managed host compared to running many ad-hoc commands.

The tasks in Ansible playbooks are written in Yet Another Markup Language (YAML), which is easier to understand than a JSON or XML file. Each task in the playbook is executed sequentially for each host in the inventory file before moving on to the next task.

Let's create a simple Ansible playbook example that will install Nginx and a MySQL server on the managed hosts that we had already defined in the inventory file.

To be more precise, we want Nginx installed on hosts in the `webservers` group and a MySQL server installed on hosts in the `dbservers` group.

```
$ vi playbook.yml

---
- hosts: webservers
  gather_facts: yes
  become_user: root
  tasks:
  - name: Install Nginx
    apt: pkg=nginx state=present
    notify:
    - restart nginx
  - name: Enable Nginx during boot
    service: name=nginx state=started enabled=yes
  handlers:
    - name: restart nginx
      service: name=nginx state=restarted

- hosts: dbservers
  become_user: root
  tasks:
  - name: Install mysql
    apt: pkg=mysql-server state=present
```

The `hosts` tells Ansible on which hosts to run the tasks. The above Ansible playbook includes two host groups from the inventory file. The tasks for `webservers` group are to *install Nginx* and *enable Nginx during boot*, and the `dbservers` group includes a single task to *install MySQL*.

The `become_user` in both the host section tells ansible to use `sudo` to run the tasks.

The `gather_facts` option gathers information about managed hosts such as distribution, OS family, and more. In ansible terminology, this information is known as FACTS.

The `handlers` section restarts Nginx when Ansible gets notified that Nginx has been installed.

A handler is the same as a task, but it will be executed when called by another task. It is like an event-driven system. A handler will run a task only when it is called by an event it listens for.

Now run the above playbook example using `ansible-playbook`. Append the name of the user from a remote managed host in the command using "-u" switch.

```
$ ansible-playbook playbook.yml -u USER

PLAY [webservers]
*********************************************************************
***********************************

TASK [Gathering Facts]
*********************************************************************
*******************************
ok: [123.45.67.89]

TASK [Install Nginx]
*********************************************************************
*********************************
changed: [123.45.67.89]

TASK [Enable Nginx during boot]
*********************************************************************
**
ok: [123.45.67.89]

RUNNING HANDLER [restart nginx]
*********************************************************************
***********************
changed: [123.45.67.89]

PLAY [dbservers]
*********************************************************************

TASK [Gathering Facts]
*********************************************************************
*******************************
ok: [123.45.67.89]

TASK [Install mysql]
*********************************************************************
*********************************
changed: [123.45.67.89]

PLAY RECAP
*********************************************************************
*****************************************
123.45.67.89               : ok=6    changed=3    unreachable=0
failed=0
```

The last line contains information about the current run of the above playbook. The four points of data are:

- `ok`: The number of tasks that were either executed correctly or didn't result in a change.
- `changed`: The number of things that were modified by Ansible.

- `unreachable`: The number of hosts that were unreachable for some reason.
- `failed`: The number of tasks failed to execute correctly.

Roles

In Ansible, a **role** provides a mechanism to break a complicated playbook into multiple reusable components. Each component offers a small function that can be used independently within the playbook. So rather than creating one complex playbook, you can create many roles and simply drop them into your playbooks.

You can't execute roles directly, the way you do a playbook, and you can't specify which host you want to execute a role, the way you would an ad-hoc command. Instead, they're built into the playbooks you use to define a host.

The Ansible Galaxy repository has thousands of pre-built roles for you to choose from, although you're free to create your role framework. Let's dig into how you might want to do just that.

Variables

In Ansible, **variables** are similar to variables in any programming language—they let you input values and numbers dynamically into your playbook. Variables simplify operations by allowing you define and declare them throughout all the various roles and tasks you want to perform.

There are few places where you can define variables in an Ansible playbook.

- In the playbook
- In the inventory file
- In a separate variable file
- Using `group_vars`

To define variables in a playbook, use `vars` key just above the task where you want to use the variable. Once declared, you can use it inside the `{{ }}` tag. Let's declare a variable by the name `pkgname` and assign it the value of the package name that we want to install, which is `nginx`. Once done, we can use the variable in a task.

```
---
- hosts: webservers
  gather_facts: yes
  become_user: root

  vars:
    pkgname: nginx

  tasks:
  - name: Install "{{ pkgname }}"
    apt: pkg="{{ pkgname }}" state=present
    ...
    ...
```

It is also possible to declare a variable in the inventory file using the syntax `[host_group_name:vars]`. Let's define the variable `pkgname` in the inventory file.

```
[webservers:vars]
pkgname=nginx
```

Now the variable `pkgname` can be used anywhere in the `webservers` hosts section in the playbook.

You can also define variables in a separate variable file and import it into the playbook. Create a variable file using `vi` another text editor and define the variable `pkgname` here.

```
$ vi ansible_vars.yml

---
pkgname: nginx
```

To use the variable `pkgname`, import the above file using the `vars_files` keyword in the playbook.

```
$ vi playbook.yml

---
- hosts: webservers
  gather_facts: yes
  become_user: root

  vars_files:
    - ./ansible_vars.yml
...
...
```

Another preferred way of managing variables is to create a `group_vars` directory inside your Ansible working directory. Ansible will load any YAML files in this directory with the name of any Ansible group.

Create the directory `group_vars` in your Ansible working directory, and then create the variable files matching with the group name from the inventory file. In our example, this would be `webservers` and `dbservers`. This allows you to separate variables according to host groups, which can make everything easier to manage.

```
$ cd <your_ansible_working_directory>
$ mkdir group_vars
$ cd group_vars
$ vi webservers

---
pkgname: nginx

$ vi dbservers

---
pkgname: mysql-server
```

You don't need to declare the variable in your playbook, as Ansible will automatically pull the variables from each `group_vars` files and will substitute them during runtime.

Now suppose you want to have variables that will apply to all the host groups mentioned in the inventory file. To accomplish it, name a file by the name `all` inside `group_vars` directory. The `group_vars/all` files are used to set variables for every host that Ansible connects to.

Conditionals

In Ansible, **conditionals** are analogous to an `if` statement in any programming language. You use a conditional when you want to execute a task

based on certain conditions.

In our last playbook example, we installed Nginx, so let's extend that by creating a task that installs Nginx when Apache *is not* present on the host. We can add another task to the playbook we've already built.

```
...
...
  tasks:
  - name: Check if Apache is already installed
    shell: dpkg -s apache2 | grep Status
    register: apache2_is_installed
    failed_when: no
  - name: Install "{{ pkgname }}"
    apt: pkg="{{ pkgname }}" state=present
    when: apache2_is_installed.rc == 1
    notify:
    - restart nginx
...
...
```

The first task in the above playbook checks if Apache is installed using `dpkg -s` command and stores the output of the task to `apache2_is_installed` variable. The return value of the task will be a non-zero value if Apache is *not* installed on the host.

Usually, Ansible would stop executing other tasks because of this non-zero value, but the `failed_when: no` gives Ansible permission to continue with the next set of tasks when it encounters a non-zero value.

The second task will install Nginx only when the return value of `rc` is equal to one, which is declared via `when: apache2_is_installed.rc == 1`.

Loops

All programming languages provide a way to iterate over data to perform some repetitive task. Ansible also provides a way to do the same using a concept called **looping**, which is supplied by Ansible lookup plugins. With loops, a single task in one playbook can be used to create multiple users, install many packages, and more.

While there are many ways to use loops in Ansible, we'll cover just one of them to get you started. The easiest way to use loops in ansible is to use `with_items` keyword, which is used to iterate over an item list to perform some repetitive tasks. The following playbook includes a task which installs packages in a loop using the keyword `with_items`.

```
---
- hosts: webservers
  gather_facts: yes
  become_user: root

  tasks:

  - name: Installing packages using loops
    apt: pkg={{ item }} state=present update_cache=yes
    with_items:
      - sysstat
      - htop
      - git
```

Run the above playbook from your command line, and you'll see that you've installed all three packages on the remote host with a single task!

Tags

Tags allow you to run only specific tasks from your playbook via the command line. Just add the `tags` keyword for each task and run only the task(s) that you want by using

- -tags

switch at the end of the ansible command. In the following playbook, we have added tags at the end of each task, thereby allowing us to run tasks separately from a single playbook.

```
---
- hosts: webservers
  gather_facts: yes
  become_user: root

  tasks:
  - name: Check if Apache is already installed
    shell: dpkg -s apache2 | grep Status
    register: apache2_is_installed
    failed_when: no
  - name: Install "{{ pkgname }}"
    apt: pkg="{{ pkgname }}" state=present
    when: apache2_is_installed.rc == 1
    notify:
    - restart nginx
  - name: ensure nginx is running and enable it at boot
    service: name=nginx state=started enabled=yes
    tags:
    - mytag1

  handlers:
    - name: restart nginx
      service: name=nginx state=restarted

- hosts: dbservers
  become_user: root
  tasks:
  - name: Install mysql
    apt: pkg="{{ pkgname }}" state=present
    tags:
    - mytag2
```

Now run any of the tasks by specifying tag name at the end of ansible command.

```
$ ansible-playbook playbook.yml -u ansadm --tags 'mytag2'
```

How to use Ansible templates

Typically, after installing a web server like Nginx, you need to configure a virtual hosts file to properly serve a given website on your VPS. Instead of using SSH to log into your VPS to configure it *after* running Ansible, or using Ansible's `copy` module to copy many unique configuration files individually, you can take advantage of Ansible's **templates** features.

A template file contains all of the configuration parameters you need, such as the Nginx virtual host settings, and uses variables, which are replaced by the appropriate values when the playbook is executed. Template files usually end with the .j2 extension that denotes the Jinja2 templating engine.

To begin working with templates, create a directory for template files in your Ansible working directory.

```
$ mkdir templates
```

Create two template files. The first template file will be the default `index.html` file for each site, and the second template file will contain configuration settings for the Nginx virtual host.

```
$ cd templates
$ vi index.html.j2
<html>
You are visiting {{ domain_name }} !
</html>
```

Similarly, create a template file for the Nginx virtual host:

```
$ vi nginx-vh.j2
server {
        listen       80;
        server_name  {{ domain_name }};
        client_max_body_size 20m;
        index index.php index.html index.htm;
        root   /var/www/html/{{ domain_name }};

        location / {
                    try_files $uri $uri/ /index.html?q=$uri&$args;
        }
        location ~* \.(js|css|png|jpg|jpeg|gif|ico|woff|ttf|svg|otf)$ {
               expires 30d;
               add_header Pragma public;
               add_header Cache-Control "public";
               access_log off;
    }
}
```

Notice that the variables `domain_name` in the above two template files are enclosed within `{{ }}`, which means they will be substituted during runtime by the value of this variable. To define the variable `domain_name`, navigate to the `group_vars` directory and edit the file `webservers` a nd add the following lines in it.

```
$ cd group_vars
$ vi webservers

---
domain_name: SUBDOMAIN.DOMAIN.TLD
```

Finally, edit the ansible playbook to create a root folder for sites, copy the `index.html` file to the site's root folder, and copy the virtual host file to the Nginx virtual host directory `/etc/nginx/sites-enabled` one by one.

```
$ vi playbook.yml

---
- hosts: webservers
  gather_facts: yes
  become_user: root

  tasks:
  - name: Check if Apache is already installed
    shell: dpkg -s apache2 | grep Status
    register: apache2_is_installed
    failed_when: no

  - name: Install "{{ pkgname }}"
    apt: pkg="{{ pkgname }}" state=present
    when: apache2_is_installed.rc == 1
    notify:
    - restart nginx

  - name: ensure nginx is running and enable it at boot
    service: name=nginx state=started enabled=yes

  - name: create virtual host root directory
    file: name=/var/www/html/{{ domain_name }} state=directory

  - name: Copying index file to webroot
    template:
      src: templates/index.html.j2
      dest: /var/www/html/{{ domain_name }}/index.html

  - name: Enables nginx virtual host
    template:
      src: templates/nginx-vh.j2
      dest: /etc/nginx/sites-enabled/{{ domain_name }}

  - name: restart nginx
    service: name=nginx state=restarted
```

```
    tags:
    - mytag1

  handlers:
    - name: restart nginx
      service: name=nginx state=restarted

- hosts: dbservers
  become_user: root
  tasks:
  - name: Install mysql
    apt: pkg="{{ pkgname }}" state=present
```

```
    tags:
    - mytag2
```

The template task in the above Ansible playbook takes two mandatory parameters `src` and `dest`. There are also a few optional parameters that can be specified in a template task but is not required at this stage.

- The `src` parameter specifies the name of the template file from templates directory that Ansible will copy to the remote server. In our case, the two templates files that we have created are `index.html.j2` and `nginx-vh.j2`
- The `dest` parameter is the path in the remote server where the file should be placed.

Finally, run the playbook from your ansible working directory:

```
$ ansible-playbook playbook.yml -u USER
```

Ansible Vault

As Ansible is being used for automation, there is a high possibility that playbooks contain certain credentials, SSL certificates or other sensitive data. Saving such sensitive data as plain text is a bad idea. One wrong commit to GitHub or laptop theft can cause an organization a huge loss. This is where Ansible vault comes into the picture. It is a great way of having infrastructure as code, without compromising on the security.

Suppose, we have a playbook that provisions your EC2 instance on AWS. You need to provide your AWS access key id and AWS secret key in the playbook. You do not share these keys with others for obvious reasons. How do you keep them unexposed? There are two ways – Either encrypt these two variables and embed them into the playbook or encrypt the entire playbook.

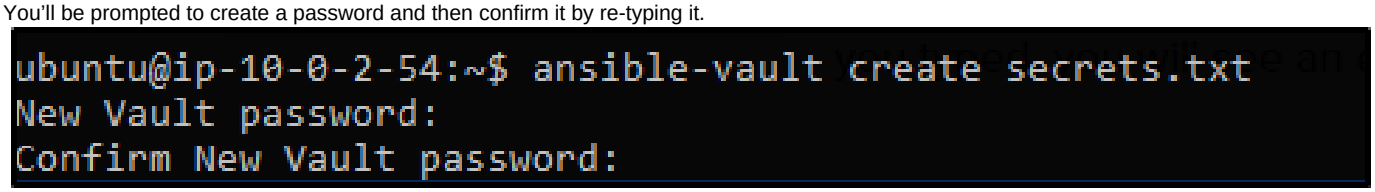
This was just one of the scenario where ansible vault can be used. We can either encrypt entire files or just encrypt few variables which might hold sensitive data and then Ansible automatically decrypts them during runtime. Now we can safely commit these values to GitHub.

Creating Encrypted File

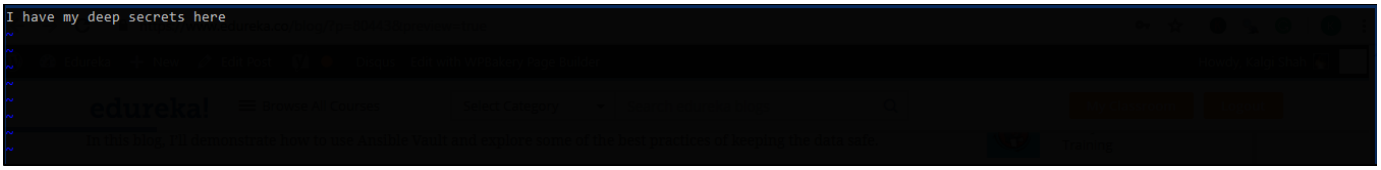
To create an encrypted file, use the **ansible-vault create** command and pass the filename.

```
$ ansible-vault create filename.yaml
```

You'll be prompted to create a password and then confirm it by re-typing it.



Once your password is confirmed, a new file will be created and will open an editing window. By default, the editor for Ansible Vault is vi. You can add data, save and exit.



And your file is encrypted.

```
ubuntu@ip-10-0-2-54:~$ cat secrets.txt
$ANSIBLE_VAULT;1.1;AES256
64633735613636656665343436316337626635316161323130323236343039303935656132613937
30313536643466356134316166316637313133393462313390a343233666163633433613232613631
32353163313033323739656664376536333038633038326639653738333435383961666233333661
36333630373665336 10a6635656462303532393534623333386231643933613864313163303439 62
65643839663737623134306366653239626636303866323030323634656365306165373730353935
31333465323839656564633464663962386666663130373032396363323863633936316264663439
65376432373165396465333236656 4633739
```

Editing Encrypted Files

If you want to edit an encrypted file, you can edit it using **ansible-vault edit** command.

```
$ ansible-vault edit secrets.txt
```

Where secrets.txt is an already created, encrypted file.

You'll be prompted to insert the vault password. The file(decrypted version) will open in a vi editor and then you can make the required changes.

```
ubuntu@ip-10-0-2-54:~$ ansible-vault edit secrets.txt
Vault password:
```

If you check the output, you'll see your text will be encrypted automatically when you save and close.

```
ubuntu@ip-10-0-2-54:~$ cat secrets.txt
$ANSIBLE_VAULT;1.1;AES256
64633735613636656665343436316337626635316161323130323236343039303935656132613937
30313536643466356134316166316637313133393462313390a343233666163633433613232613631
32353163313033323739656664376536333038633038326639653738333435383961666233333661
36333630373665336 10a6635656462303532393534623333386231643933613864313163303439 62
65643839663737623134306366653239626636303866323030323634656365306165373730353935
31333465323839656564633464663962386666663130373032396363323863633936316264663439
65376432373165396465333236656 4633739
```

Viewing Encrypted File

If you wish to just view an encrypted file, you can use the **ansible-vault view** command.

```
$ ansible-vault view filename.yml
```

Again you'll be prompted for a password.

```
ubuntu@ip-10-0-2-54:~$ ansible-vault view secrets.txt
Vault password:
```

and you'll see similar output.

```
ubuntu@ip-10-0-2-54:~$ ansible-vault view secrets.txt
Vault password:
I have my deep secrets here
Adding more secrets
```

Rekeying Vault Password

Of course, there are times where you'll want to change the vault password. You can use the **ansible-vault rekey** command.

```
$ ansible-vault rekey secrets.txt
```

You'll be prompted with the vault's current password and then the new password and finally done by confirming the new password.

```
ubuntu@ip-10-0-2-54:~$ ansible-vault rekey secrets.txt
Vault password:
New Vault password:
Confirm New Vault password:
Rekey successful
ubuntu@ip-10-0-2-54:~$
```

Encrypting Unencrypted Files

Suppose you have a file which you wish to encrypt, you can use the **ansible-vault encrypt** command.

```
$ ansible-vault encrypt filename.txt
```

You'll be prompted to insert and confirm password and your file is encrypted.

```
ubuntu@ip-10-0-2-54:~$ sudo ansible-vault encrypt credentials.txt
New Vault password:
Confirm New Vault password:
Encryption successful
ubuntu@ip-10-0-2-54:~$
```

Now that you look at file contents, its all encrypted.

```
ubuntu@ip-10-0-2-54:~$ sudo cat credentials.txt
$ANSIBLE_VAULT;1.1;AES256
35633731366336363332383238323339626461373230306131396631383765333335626133616237
64623362343161663433346236303262643231376231363330a306332653634613735646631613034
32366432346439663337303966646539656639323965393462626135633864386330653433396634
653266363135336266 0a636631316435396236613365363435613630386461333863646666656564
66663365636336656638376537663230373764336364323136636333623661663662303561373730
633364383539313865616134326261363365616539363030393539
ubuntu@ip-10-0-2-54:~$
```

Decrypting Encrypted Files

If you want to decrypt an encrypted file, you can use **ansible-vault decrypt** command.

```
$ ansible-vault decrypt filename.txt
```

As usual, it'll prompt you to insert and confirm the vault password.

```
ubuntu@ip-10-0-2-54:~$ sudo ansible-vault decrypt credentials.txt
Vault password:
Decryption successful
ubuntu@ip-10-0-2-54:~$
```

Encrypting specific variables

Best practice while using Ansible Vault is to encrypt only the sensitive data. In the example explained above, the development team does not want to share their password with the production and the staging team but they might need access to certain data to carry out their own task. In such cases you should only be encrypting the data you do not want to share with others, leaving the rest as it is.

Ansible Vault allows you to encrypt only specific variables. You can use the **ansible-vault encrypt_string** command for this.

```
$ ansible-vault encrypt_string
```

You'll be prompted to insert and then confirm the vault password. You can then start inserting the string value that you wish to encrypt. Press ctrl-d to end input. Now you can assign this encrypted value to a string in the playbook.

```
ubuntu@ip-10-0-2-54:~$ ansible-vault encrypt_string
New Vault password:
Confirm New Vault password:
Reading plaintext input from stdin. (ctrl-d to end input)
password!vault |
          $ANSIBLE_VAULT;1.1;AES256
          6339343965346639316664623562313937626238346437353938363566376362656261353365663 6
          3237633832613065316435033663139663039656464373 20a6137366433356236636435626630 65
          3164633465343932383861336337623133623537613134393636333531383037356661333066373 1
          3732626436666461330a3231633135613436646432326239393363393733383038666643734396131
          6332
Encryption successful
ubuntu@ip-10-0-2-54:~$
```

You can also achieve the same thing in a single line.

```
$ ansible-vault encrypt_string 'string' --name 'variable_name'
```

```
ubuntu@ip-10-0-2-54:~$ ansible-vault encrypt_string 'all my password' --name 'password'
New Vault password:
Confirm New Vault password:
password: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          3738633637623733303466643263356435383464373865306437383935313736633763623662663 9
          3236356638363062303932386639333539356334623830620a653163653439633538396131636235
          6665323161333364383164303063353533383161363163373363646239383739666432326231663 9
          6331656130363336300a373239626662303935393963666634363463346346165646162303439336464
          3136
Encryption successful
ubuntu@ip-10-0-2-54:~$
```

Decrypting Encrypted Files During Runtime

If you wish to decrypt a file during runtime, you could use **–ask-vault-pass** flag.

```
$ ansible-playbook launch.yml --ask-vault-pass
```

This will decrypt all the encrypted files that are used for this launch.yml playbook to execute. Also, this is only possible if all the files are encrypted with the same password.

```
ubuntu@ip-10-0-2-54:~$ sudo ansible-playbook launch.yml --ask-vault-pass
Vault password:
```

Password prompts can get annoying. The purpose of automation becomes pointless. How do we make this better? Ansible has a feature called “password file” which references to a file containing the password. You can then just pass this password file during runtime to automate it.

```
$ ansible-playbook launch.yml --vault-password-file ~/ .vault_pass.txt
```

```
ubuntu@ip-10-0-2-54:~$ sudo ansible-playbook launch.yml --vault-password-file /home/ubuntu/vault_pass.txt
```

Having a separate script that specifies the passwords is also possible. You need to make sure the script file is executable and the password is printed to standard output for it to work without annoying errors.

```
$ ansible-playbook launch.yml --vault-password-file ~/ .vault_pass.py
```

Using Vault Id

Vault Id is a way of providing an identifier to a particular vault password. Vault ID helps in encrypting different files with different passwords to be referenced inside a playbook. This feature of Ansible came out with the release of Ansible 2.4. Prior to this release, only one vault password could be used in each ansible playbook execution.

So now if you wish to execute an Ansible playbook which uses multiple files encrypted with different passwords, you can use Vault Id.

```
$ ansible-playbook --vault-id vault-pass1 --vault-id vault-pass2
filename.yml
```

With this, we come to the end of this Ansible Vault blog. It is amazing to catch up with technology and make the fullest of them but not by compromising on security. This is one of the best ways to have Infrastructure as code(IaC).