Ansible:

Overview

- **Ansible** is open source software that automates software provisioning, configuration management, and application deployment.
- **Ansible** connects via SSH, remote PowerShell and other remote APIs.
- Some of **Ansible** design principals are:
    - Simple setup process and a minimal learning curve
    - Manage machines very quickly and in parallel
    - Avoid custom-agents and additional open ports
    - Describe infrastructure in a language that is both machine and human friendly
    - Be the easiest IT automation system to use, ever
- For example, let's say we have 10 servers, which all need to have Nginx up and running.
- The "old" process will be: SSH into the machine → apt-get install Nginx → start Nginx **X 10**

https://docs.ansible.com/

Setup

- Ansible uses python interpreter to run its modules, therefore we will need to install Python on the host in order for Ansible to communicate with it.

```
$ sudo apt-get update

$ sudo apt-get install python
```

- Next we will install Ansible

```
$ sudo apt update

$ sudo apt install software-properties-common

$ sudo apt update

$ sudo apt install ansible
```

- To check Ansible is installed correctly, let’s run the following command:

```
$ ansible –version
```

- Install sshpass which allows us to provide the ssh password without using the prompt:

```
$ sudo apt-get install sshpass
```

Host / Inventory

- Ansible keeps track of all of the servers that it knows about through a hosts file (also known as inventory file).
- We need to set up this file first before we can begin to communicate with our other computers (servers).

```
$ sudo nano /etc/ansible/hosts
```

- For our practice, Let’s start a VM, and obtain its IP address (using ip a)
- Let’s add our server information including IP address, privileged user name, and user password one under the other and save:

```
10.0.1.2 ansible_ssh_pass=123456 ansible_ssh_user=root
```

- Let’s ping our servers, to validate connection:

```
$ ansible -m ping all
```

Host group

- Ansible lets us use host groups which are used in classifying systems and deciding what systems you are controlling at what times and for what purpose.
- For example:

```
[webservers]
10.0.1.2 ansible_ssh_pass=123456 ansible_ssh_user=root
 [dbservers]
10.2.8.0 ansible_ssh_pass=123456 ansible_ssh_user=root
```

- Using it like below:

```
$ ansible -m ping webservers
```

Variables definition

Earlier we saw how we can create host groups:

```
[webservers]

    52.173.14.216
    ansible_ssh_private_key_file=/home/user/456.pem
    ansible_ssh_user=ubuntu


    55.168.16.276
    ansible_ssh_private_key_file=/home/user/789.pem
    ansible_ssh_user=ubuntu
```

On top of it we can add a variable name to our host:

```
Server1 52.173.14.216
ansible_ssh_private_key_file=/home/user/456.pem
ansible_ssh_user=ubuntu

Server2 55.168.16.276
ansible_ssh_private_key_file=/home/user/789.pem
ansible_ssh_user=ubuntu
```

The last layer can be grouping the variables, which will help us mix and match:

```
[us-east]
Server1
Server2
```

Terminology

- Modules:
  - Modules are discrete units of code that can be used from the command line or in a playbook task.
  - Each module supports taking arguments. Nearly all modules take key=value arguments, space delimited.
- Tasks:
  - Each play contains a list of tasks. Tasks are executed in order, one at a time, against all machines matched by the host pattern, before moving on to the next task. It is important to understand that, within a play, all hosts are going to get the same task directives.
  - The goal of each task is to execute a module, with very specific arguments. Variables, as mentioned above, can be used in arguments to modules.
- Handlers:
  - The easiest way to think about Handlers is that they are fancy Tasks.
  - Tasks are Ansible's way of doing something and Handlers are our way of calling a Task after some other Task completes.
  - For example, we will want to configure security options to a server after installing it.

Playbook

- Playbooks allow you to organize your configuration and management tasks in simple, human-readable files.
- Each playbook contains a list of tasks (‘plays’ in Ansible parlance) and are defined in a YAML file.
- Playbooks and roles are large topics so I encourage you to read the docs.
- Let’s create a simple example playbook, which will ping all servers:

```
$ sudo nano ping.yml
```

- containing the following:

```
- hosts:
    - all
  tasks:
    - action: ping
```

- To run it simply run:

```
$ ansible-playbook ping.yml
```

- ** Remember yml files can be annoying, so I suggest using http://www.yamllint.com

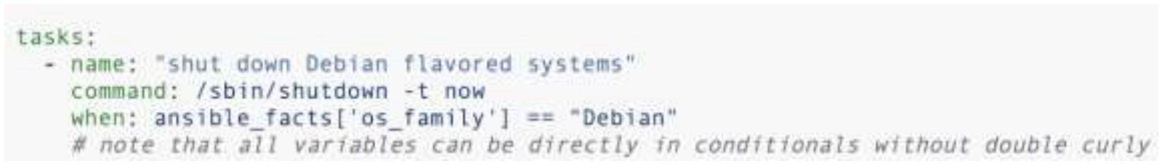
Dry mode

- Earlier we saw how we can run our playbooks on our hosts.
- Another feature Ansible brings is called “Dry mode” (also called Check Mode), in this mode, Ansible will not make any changes to your host, but simply report what changes would have been made if the playbook was run without this flag.
- This is achived by using the **--check** flag

```
$ sudo ansible-playbook --check ntest.yml
```

Conditions

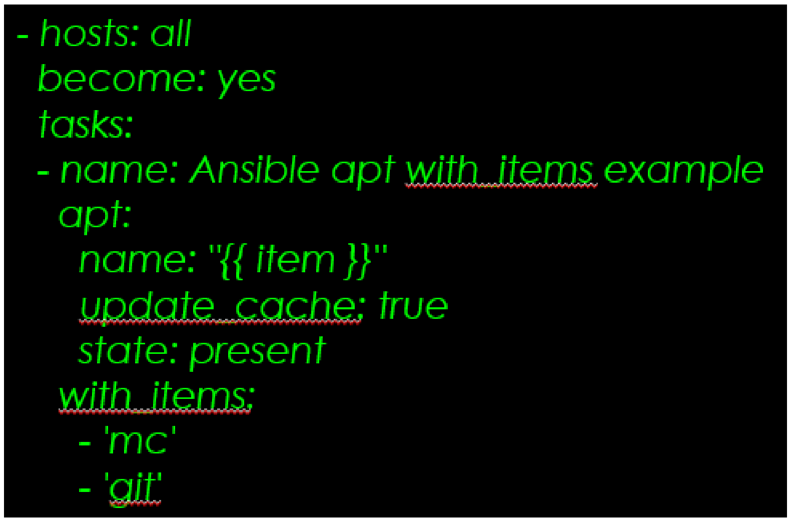
- Sometimes we will want to skip a particular step on a particular host.
- This could be something as simple as not installing a certain package if the operating system is a particular version, or it could be something like performing some cleanup steps if a filesystem is getting full.



```
tasks:
  - name: "shut down Debian flavored systems"
    command: /sbin/shutdown -t now
    when: ansible_facts['os_family'] == "Debian"
    # note that all variables can be directly in conditionals without double curly
```

Loops

- Loops can be used to shorten processes.
- We can use the with_items syntax:



```
- hosts: all
  become: yes
  tasks:
  - name: Ansible apt with_items example
    apt:
      name: "{{ item }}"
      update_cache: true
      state: present
    with_items:
      - 'mc'
      - 'git'
```

Loops

- Roles are good for organizing multiple, related Tasks and encapsulating data needed to accomplish those Tasks.
- For example, installing Nginx may involve adding a package repository, installing the package, and setting up configuration.
- We've seen installation via a Playbook, but once we start configuring our installations, the Playbooks tend to get a little busier.
- Furtermore, real-world configuration often requires extra data such as variables, files, dynamic templates and more.
- These tools can be used with Playbooks, but we can do better immediately by organizing related Tasks and data into one coherent structure: a Role.

- Roles expect files to be in certain directory names.
- Roles must include at least one of the below directories:
    - tasks - contains the main list of tasks to be executed by the role.
    - handlers - contains handlers, which may be used by this role.
    - defaults - default variables for the role.
- When in use, each directory must contain a main.yml file, which contains the relevant content.
- Create a new role named **nginx** using ansible-galaxy command, which will create a roles directory with the new role name and its sub directories:

```
$ ansible-galaxy init nginx
```

- Inside **roles/nginx/tasks** we will define the following tasks
    - Add Nginx repository
    - Install Nginx
- Inside **roles/nginx/handlers** we will define the handlers which will start/reload Nginx once installed.
- **The files inside sample folder can be used.**
- Create a new playbook **server.yaml** with the following content:

```
---
- hosts: local
  connection: local
  roles:
    - nginx
```

- Run the following command:

```
$ sudo ansible-playbook -i hosts server.yml
```

Ansible vault

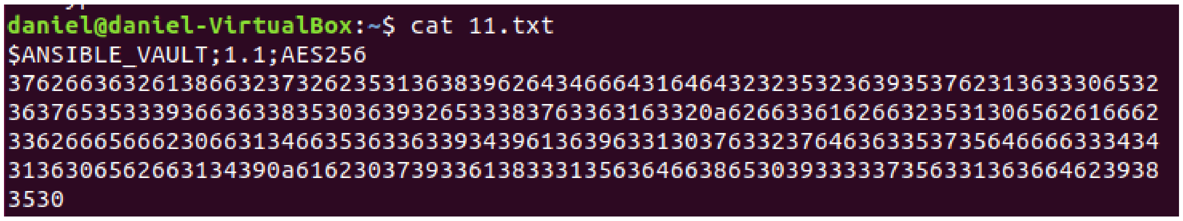
- If one of your tasks requires sensitive information (for example: database user and password), it's a good practice to keep this information encrypted, instead of storing it in plain text.
- Ansible ships with a command line tool called ansible-vault, that allows you to create and manage encrypted files.
- This way you can commit the encrypted file to your source control and only users with the decryption password will be able to read it.
- To encrypt a file which contains text (any!), use:

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

- Now, let's try to print its content using **cat**:

```
$ cat secrets.txt
```

- The output will be encrypted, and will look like that:



```
daniel@daniel-VirtualBox:~$ cat 11.txt
$ANSIBLE_VAULT;1.1;AES256
37626636326138663237326235313638396264346664316464323235323639353762313633306532
3637653533393663633835303639326533383763363163320a626633616266323531306562616662
33626665666230663134663536336363934396136396331303763323764636335373564666333434
3136306562663134390a616230373933613833313563646638653039333337356331363664623938
3530
```

- To decrypt the file, use **decrypt** command:

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

- To edit the encrypted file, simply, use **edit** command:

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

- To change encryption password, simply use **rekey** command:

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

- To view encrypted file content, simply, use **view** command:

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

Ansible tower

- One of the major gripes from Ansible users is that it didn't have a proper GUI.
- And that's putting it mildly--the GUI was so bad that in the early days it wasn't even properly synced to the CLI, meaning that the CLI and GUI could give you 2 different query results about the state of a certain node.
- Ansible itself was (and still is) rather new, so most of its users were by definition new users.
- Ansible Tower, previously called the AWX project, is the fix to this problem.
- It is a web-based UI for Ansible, containing the most important Ansible features, especially those that render better as graphical rather than text-based output.
- Ansible tower is not free, and the base plans are starting from 5,000-17,500$/year.