**Ansible:-**

* **Introduction**
* **Env Setup**
* **Ad hoc Commands**
* **Inventory**
* **Add Remote Servers to Inventory**
* **Modules**
* **Setup – gather facts about remote hosts**
* **Playbooks**
* **Variables**
* **Ansible - conditional statements**
* **Ansible - Loops & Arrays**
* **Handlers**
* **Tags**
* **Templates**
* **Roles**
* **Ansible Vault – Managing secrets with Ansible Vault**
* **Ansible – Debug the Playbook**
* **Ansible Galaxy**

**Ansible – Introduction**

Ansible is an IT automation tool. Ansible is simple open source IT engine which automates application deployment, intra service orchestration, cloud provisioning and many other IT tools.

Ansible is a configuration management and provisioning tool, similar to Chef, Puppet or Salt.

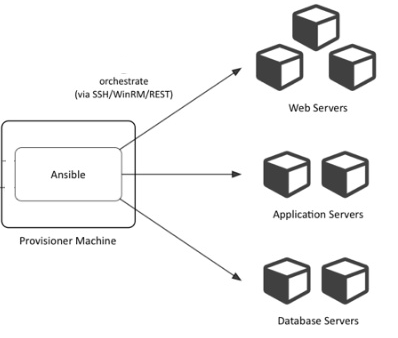
Ansible’s main goals are simplicity and ease-of-use. It also has a strong focus on security and reliability, featuring a minimum of moving parts, usage of OpenSSH for transport (with other transports and pull modes as alternatives), and a language that is designed around auditability by humans–even those not familiar with the program.

One of the simplest and the easiest to get started with. It uses SSH to connect to servers and run the configured Tasks.

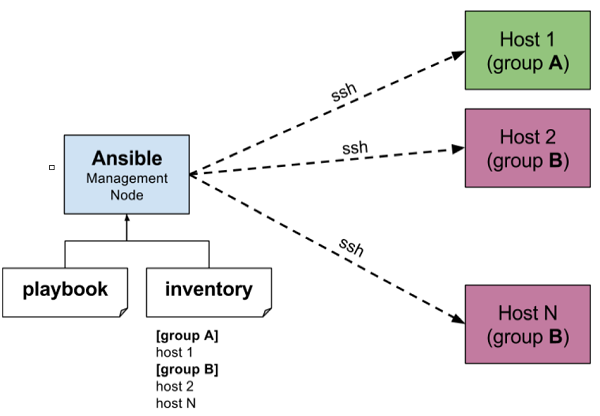
One nice thing about Ansible is that it's very easy to convert bash scripts (still a popular way to accomplish configuration management) into Ansible Tasks. Since it's primarily SSH based, it's not hard to see why this might be the case - Ansible ends up running the same commands.

We could just script our own provisioners, but Ansible is much cleaner because it automates the process of getting context before running Tasks. With this context, Ansible is able to handle most edge cases - the kind we usually take care of with longer and increasingly complex scripts.

**Ansible Architecture**



**How Ansible works?**



**Ansible Tasks are idempotent.** Without a lot of extra coding, bash scripts are usually not safety run again and again. Ansible uses "Facts", which is system and environment information it gathers ("context") before running Tasks.

Ansible uses these facts to check state and see if it needs to change anything in order to get the desired outcome. This makes it safe to run Ansible Tasks against a server over and over again.

**Monitor the Configuration regularly on hosts:**

Write all config/deployments in Playbook and set cronjob or config it in Jenkins.

**Ansible - Env setup**

**Pre-requisities:**

**Install Python in all servers**

Sudo apt-get install python

**Installation:**

sudo apt-get update

sudo apt-get install -y ansible

**Ansible Home Dir:** /etc/ansible

ubuntu@ip-172-31-20-78:/etc/ansible$ ls

ansible.cfg hosts

**Ansible - Ad hoc Commands**

Adhoc commands are simple one line command to perform some action. Running modules with Ansible commands are adhoc commands.

These ad-hoc commands are not used for configuration management and deployment, because these commands are of one time usage.

ansible-playbook is used for configuration management and deployment.

**Examples:**

**Display Path Variable of servers**

$ ansible all -m shell -a 'echo $PATH'

**Creating new directory in servers**

$ ansible all -m file -a "dest=/home/myuser2/new state=directory"

**Deleting whole directory and files**

$ ansible all -m file -a "dest=/path/user1/new state=absent"

**Ansible – Inventory**

Ansible works against multiple systems in your infrastructure at the same time. It does this by selecting portions of systems listed in Ansible’s inventory, which defaults to being saved in the location /etc/ansible/hosts. You can specify a different inventory file using the -i <path> option on the command line.

**Example:**

mail.example.com

[webservers]

foo.example.com

bar.example.com

[dbservers]

one.example.com

two.example.com

three.example.com

**Custom Inventory:**

Ex: ansible-playbook -i /etc/ansible/hostsnew nginx.yml

**Add Remote Servers to Inventory:**

**At Ansibe Controller:**

**Install sshpass:**

sudo apt-get install sshpass

# uncomment this to disable SSH key host checking (ansible.cfg)

host\_key\_checking = False

**Update Inventory config:**

Remote ansible\_host=172.31.26.191 ansible\_user=myuser2 ansible\_password=myuser2 ansible\_sudo\_pass=myuser2

Local ansible\_host=172.31.20.78 ansible\_user=ubuntu ansible\_password=ubuntu ansible\_sudo\_pass=ubuntu

**At Remote servers & Controller**

Use a Match block at the end of /etc/ssh/sshd\_config:

# Global settings

PasswordAuthentication yes

Then tell the sshd service to reload its configuration:

service ssh reload

Give admin privileges to myuser2 for the below dirs

/tmp/

/home/myuser2/

chown myuser2:myuser2 /home/myuser2

chmod –R 777 /home/myuser2/

chmod –R 777 /tmp/

Add User to sudo Group

usermod -aG sudo myuser2

Run from Controller - to the check the connectivity

**ansible local -m ping**

**Ansible – Modules**

Modules (also referred to as “task plugins” or “library plugins”) are discrete units of code that can be used from the command line or in a playbook task.

Modules in Ansible are idempotent. From a RESTful service standpoint, for an operation (or service call) to be idempotent, clients can make that same call repeatedly while producing the same result. In other words, making multiple identical requests has the same effect as making a single request.

Let’s review how we execute three different modules from the command line:

Example: copy, service, ping, file …etc

$ ansible webservers -m service -a "name=nginx state=started"

$ ansible webservers -m ping

$ ansible webservers -m command -a "/sbin/reboot -t now"

$ ansible all -m file -a "dest=/home/ubuntu/new state=directory"

**Setup – gather facts about remote hosts**

**$ ansible all –m setup**

**Ansible – Playbooks**

Ansible playbooks are a way to send commands to remote computers in a scripted way. Instead of using Ansible commands individually to remotely configure computers from the command line, you can configure entire complex environments by passing a script to one or more systems.

Each playbook contains one or more plays, which map hosts to a certain function. Ansible does this through something called tasks, which are basically module calls.

Ansible playbooks are written in the YAML data serialization format. JSON is another popular data serialization format, but YAML is much easier to read.

**Exploring a Basic Playbook**

Let's look at a basic playbook:

**Example 1: Install httpd**

Vi apache.yml

-----------------------------------------------------------------------------

---

- hosts: all

become: yes

tasks:

- name: Installs apache2 web server

apt: pkg=apache2 state=present update\_cache=true

notify:

- start apache2

handlers:

- name: start apache2

service: name=apache2 state=started

----------------------------------------------------------------------------------

**Example 2: Copy files**

Vi copy.yml

------------------------------------------------------------------------------------

---

- hosts: all

become: yes

tasks:

- name: Installs nginx web server

apt: pkg=apache2 state=installed update\_cache=true

notify:

- start nginx

- name: Upload default index.html for host

copy: src=source/index.html dest=/tmp mode=0644

handlers:

- name: start nginx

service: name=apache2 state=started

**Ansible - Variables**

Variable in playbooks are **very similar** to using variables in any programming language. It helps you to use and assign a value to a variable and use that anywhere in the playbook.

**Examples:**

sudo vim vim-install.yml

---

- hosts: all

become: yes

vars:

package: vim

tasks:

- name: Install Package

apt: name={{ package }} state=latest

**Run command**: ansible-playbook vim-install.yml -vvvv

**Ansible - Conditional statements**

---

- hosts: all

become: yes

tasks:

- name: Check if PHP is installed

register: php\_installed

command: php -v

ignore\_errors: true

- name: This task is only executed if PHP is installed

debug: msg='php is installed'

when: php\_installed|success

- name: This task is only executed if PHP is NOT installed

debug: msg='PHP is NOT installed'

when: php\_installed|failed

**Ansible - Loops & Arrays**

**Example 1:**

Vi addusers.yml

---

- hosts: all

become: yes

tasks:

- name: add several users

user:

name: "{{ item }}"

state: present

groups: "wheel"

with\_items:

- testuser1

- testuser2

**Run:** ansible-playbook addusers.yml

**Verify:** groups testuser1

**Example 2:**

**Vim array.yml**

- hosts: all

vars:

hello: [Asia, Americas, Artic]

tasks:

- name: Ansible array variables example

debug:

msg: "{{ item }}"

with\_items:

- "{{ hello }}"

- name: Ansible List variable Example

debug:

msg: "{{ hello[2] }}"

**Ansible – Handlers**

Handlers are just like tasks, but they only run when they have been told by a task that changes have occurred on the client system.

For instance in the below Nginx playbook, we have a handler here that starts the Nginx service after the package is installed. The handler is not called unless the "Installs nginx web server" task results in changes to the system, meaning that the package had to be installed and wasn't already there.

**Ansible – Tags**

Sometimes while writing the playbook we never think about dividing the playbook logically and we end up with a long playbook. But what if we can divide the long playbook into logical units. We can achieve this with tags, which divides the playbook into logical sections.

Ansible Tags are supported by tasks and play. You can use tags keyword and provide any tag to task or play. Tags are inherited down to the dependency chain which means that if you applied the tags to a role or play, all tasks associated under that will also get the same tag.

Vi add\_users\_groups.yml

------------------------------------

---

- hosts: all

become: yes

tasks:

- name: add users

user: name={{item}} state=present groups=group1

with\_items:

- user1

- user2

- user3

tags: add\_user

- name: add group

group: name={{item}} state=present

with\_items:

- group1

- group2

- group3

tags: add\_group

------------------------------------------------

**$ ansible-playbook add-users-groups.yml - - list-tags**

**$ ansible-playbook add-users-groups.yml - - tags add\_group**

**$ sudo ansible-playbook --limit Remote add-users-groups.yml --tags add\_users**

**$ sudo ansible Remote -m command -a "groups user1"**

**Ansible – Templates**

A template in Ansible is a file which contains all your configuration parameters, but the dynamic values are given as variables. During the playbook execution, depending on the conditions like which cluster you are using, the variables will be replaced with the relevant values.

At the bare minimum, you need to have two parameters when using the Ansible module.

src: the source of the template file. This can be relative or absolute path.

dest: the destination path on the remote server

In the following task. We are using the template module on the hello\_world.j2 file. Double braces cover the variables.

**Example 1:**

**---**

- hosts: all

vars:

var1: 'DevOps Gurus'

tasks:

- name: Ansible Template Example

template:

src: hello\_world.j2

dest: /tmp/hello\_world.txt

hello\_world.j2

--------------

Hi, {{ var1 }}

Output - hello\_world.txt

Hi, DevOps Gurus

**Example 2:**

Templating:

-------------

sudo vim index.html

----------------------------------------------------------------------

<html>

<head>

<title>This is a sample page</title>

</head>

<body>

<h1>Here is a heading!</h1>

<p>Here is a regular paragraph. Wow!</p>

</body>

</html>

-----------------------------------------------------------------------------------------

sudo vim vhost.tpl

-----------------------------------------------------------------------------------------

<VirtualHost \*:80>

ServerAdmin webmaster@localhost

DocumentRoot {{ doc\_root }}

<Directory {{ doc\_root }}>

AllowOverride All

Require all granted

</Directory>

</VirtualHost>

-------------------------------------------------------------------------------------------

sudo vim template.yml

--------------------------------------------------------------------------------------------

---

- hosts: all

become: true

vars:

doc\_root: /var/www/example

tasks:

- name: Update apt

apt: update\_cache=yes

- name: Install Apache

apt: name=apache2 state=latest

sudo: yes

- name: Create custom document root

file: path={{ doc\_root }} state=directory owner=www-data group=www-data

- name: Set up HTML file

copy: src=index.html dest={{ doc\_root }}/index.html owner=www-data group=www-data mode=0644

- name: Set up Apache virtual host file

template: src=vhost.tpl dest=/etc/apache2/sites-available/000-default.conf

notify: restart apache

handlers:

- name: restart apache

service: name=apache2 state=restarted

sudo: yes

**-------------------------------------------------------------------------------------------------**

**ansible-playbook template.yml –vvvvv**

**Ansible – Roles**

Roles provide a framework for fully independent, or interdependent collections of variables, tasks, files, templates, and modules.

In Ansible, the role is the primary mechanism for breaking a playbook into multiple files. This simplifies writing **complex playbooks**, and it makes them easier to reuse. The breaking of playbook allows you to logically break the playbook into reusable components.

Each role is basically limited to a particular functionality or desired output, with all the necessary steps to provide that result either within that role itself or in other roles listed as dependencies.

**Roles are not playbooks.** Roles are small functionality which can be independently used but have to be used within playbooks. There is no way to directly execute a role. Roles have no explicit setting for which host the role will apply to.

Top-level playbooks are the bridge holding the hosts from your inventory file to roles that should be applied to those hosts.

Creating a New Role

The directory structure for roles is essential to create a new role.

Role Structure

Roles have a structured layout on the file system. The default structure can be changed but for now let us stick to defaults.

Each role is a directory tree in itself. The role name is the directory name within the /roles directory.

**$ ansible-galaxy init elk**

**ELK configuration with Ansible Roles:**

https://github.com/Maddu/ansible-elk-playbook.git

**Check the status of Elasticsearch in remote host from Ansible Controller.**

$ sudo ansible <hosts> -m shell -a "service elasticsearch status"

**Ansible – Vault**

The “Vault” is a feature of Ansible that allows you to keep sensitive data such as passwords or keys in encrypted files, rather than as plaintext in playbooks or roles. These vault files can then be distributed or placed in source control.

[**What Can Be Encrypted With Vault**](https://docs.ansible.com/ansible/2.6/user_guide/vault.html#id6)

Ansible Vault can encrypt any structured data file used by Ansible. This can include “group\_vars/” or “host\_vars/” inventory variables, variables loaded by “include\_vars” or “vars\_files”, or variable files passed on the ansible-playbook command line with -e @file.yml or -e @file.json. Role variables and defaults are also included.

Ansible tasks, handlers, and so on are also data so these can be encrypted with vault as well. To hide the names of variables that you’re using, you can encrypt the task files in their entirety.

[**Creating Encrypted Files**](https://docs.ansible.com/ansible/2.6/user_guide/vault.html#id7)

To create a new encrypted data file, run the following command:

$ ansible-vault create foo.yml

First you will be prompted for a password. The password used with vault currently must be the same for all files you wish to use together at the same time.

[**Editing Encrypted Files**](https://docs.ansible.com/ansible/2.6/user_guide/vault.html#id8)

To edit an encrypted file in place, use the [ansible-vault edit](https://docs.ansible.com/ansible/2.6/cli/ansible-vault.html" \l "ansible-vault-edit) command. This command will decrypt the file to a temporary file and allow you to edit the file, saving it back when done and removing the temporary file:

$ ansible-vault edit foo.yml

[**Decrypting Encrypted Files**](https://docs.ansible.com/ansible/2.6/user_guide/vault.html#id11)

If you have existing files that you no longer want to keep encrypted, you can permanently decrypt them by running the [ansible-vault decrypt](https://docs.ansible.com/ansible/2.6/cli/ansible-vault.html" \l "ansible-vault-decrypt) command. This command will save them unencrypted to the disk, so be sure you do not want [ansible-vault edit](https://docs.ansible.com/ansible/2.6/cli/ansible-vault.html" \l "ansible-vault-edit) instead:

$ ansible-vault decrypt foo.yml

**Usage of Vault:**

1. **Encrypt vars and pwds**

**$ vi /etc/ansible/hosts**

Remote ansible\_host=172.31.26.191 ansible\_user=myuser2 ansible\_password=myuser2 ansible\_sudo\_pass=myuser2

Local ansible\_host=172.31.20.78 ansible\_user=ubuntu ansible\_password=ubuntu ansible\_sudo\_pass=ubuntu

**$ sudo ansible-vault encrypt hosts**

**$ ansible all -m ping --ask-vault-pass**

[**Using Vault in playbooks**](https://docs.ansible.com/ansible/latest/user_guide/playbooks_vault.html#id3)

[**Running a Playbook With Vault**](https://docs.ansible.com/ansible/latest/user_guide/playbooks_vault.html#id4)

**$ ansible-playbook nginx.yml --ask-vault-pass**

**Using Ansible Vault with a Password File**

If you do not wish to type in the Vault password each time you execute a task, you can add your Vault password to a file and reference the file during execution.

For example, you could put your password in a .vault\_pass file like this:

$ echo 'my\_vault\_password' > .vault\_pass

$ ansible-vault encrypt /etc/ansible/hosts --vault-password-file .vault\_pass

$ ansible-playbook nginx.yml --vault-password-file .vault\_pass

**Ansible – Debug the Playbook**

**Debugging Playbook on Run**

Debugging an ansible playbook is one of the coolest feature that ansible has introduced(in 2.1 version). While we develop any playbook sometimes we see failures and to debug we usually run the playbook again, identify the error that Ansible throws and modify the playbook and then rerun the playbook. What if your playbook takes 30 minutes to run and your play is failing for last few tasks and after debugging you will again run your playbook for almost 30 minutes. I guess that’s ideal way to debug, You can used the ansible debug strategy.

Ansible provides a debug strategy which will help to enable the debugger when a task fails. It will provide access to all features of the debugger in the context of failed task. This way if you encounter a failed task, you can set the values of the variables, update the module arguments and re-run the failed task with new arguments and variables.

To use the debug strategy in your playbook, you have to define the strategy as debug.

**Note:** Use Ansible 2.5 or later for debugging

Example 1:

Vi debug-example.yml

------------------------------------

---  
- hosts: web  
 strategy: debug

vars:  
 package: "git"  
 tasks:  
 - name: Install Git Package  
 apt:  
 name: "{{ new\_package }}"

----------------------------------------

Follow the steps for debugging:

1. task\_vars['new\_package'] = 'git'
2. r(redo)

Example 2:

Vi debug2.yml

-----------------------------------------------

---

- hosts: test

debugger: on\_failed

gather\_facts: no

vars:

var1: value1

tasks:

- name: wrong variable

ping: data={{ wrong\_var }}

-------------------------------------------------

Follow the steps for debugging:

1. task.args['data'] = '{{ var1 }}'
2. r(redo)

**Ansible – Galaxy**

*Ansible Galaxy is the Ansible’s official community hub for finding, downloading, rating, and sharing Ansible roles…*”

So, it is …

* a command line tool
* a website

for searching, installing, creating and managing roles.

$ ansible-galaxy install geerlingguy.nginx

<https://galaxy.ansible.com/>

login with github credentials.