# Ansible

## ANSIBLE

Index:

Configuration Management…………………………………………………………………………………… 4

What Is Configuration Management?...............................................................4

How Configuration Management Works?.......................................................4

Configuration Management Benefits……………………………………………………………………4

Why Configuration Management Is Important……………………………………………………5

Configuration Management Tools………………………………………………………………………5

Introduction to Ansible…………………………………………………………………………………………5

IT automation………………………………………………………………………………………………………6

Advantages of IT automation…………………………………………………………………………6

Types of IT automation…………………………………………………………………………………………6

What is Ansible? ………………………………………………………………………………………………7

What Ansible can it automate? ..........................................................................8

How ansible works?..................................................................................................9

Installation of ansible……………………………………………………………………………………10

Check ansible version command……………………………………………………………………10

Ansible Inventories………………………………………………………………………………………….. 12

Hosts and Groups:………………………………………………………………………………………..12

Ansible Static inventory……………………………………………………………………………13

Ansible inventory variable data…………………………………………………………………14

Host Variables:………………………………………………………………………………………………14

Group Variables:………………………………………………………………………………………. 14

Ansible dynamic inventory………………………………………………………………………16

Ad-hoc command………………………………………………………………………………… 17

YAML…………………………………………………………………………………………………18

Ansible Playbooks:……………………………………………………………………………19

Ansible verbosity…………………………………………………………………………………21

Gather facts: ……………………………………………………………………………………22

Upgrading all installed packages:………………………………………………………37

Creating the Ansible user:………………………………………………………………38

Install FTP server:……………………………………………………………………………39

Install webserver:………………………………………………………………………………41

Jinja2 templates:………………………………………………………………………………42

Jinja2 templates Conditionals controls:…………………………………………45

Working with inventory files:……………………………………………………………46

The basic inventory file:…………………………………………………………………47

Groups in an inventory file:……………………………………………………………48

Regular expressions in the inventory file:…………………………………49

Working with variables:……………………………………………………………………49

Host variables:………………………………………………………………………………… 50

Group variables:………………………………………………………………………………50

Variable files:……………………………………………………………………………………51

Overriding configuration parameters with an inventory file:……………51

Working with dynamic inventory:……………………………………………………52

# Ansible

Setting up EC2 External Inventory Script With Ansible:……………………………………53

Working Ansible loops:……………………………………………………………………………………………54

Standard loops - with_items:…………………………………………………………………………………55

Nested loops - with_nested:…………………………………………………………………………………56

Delegating a task:………………………………………………………………………………………………………57

Working with ansible conditions: when……………………………………………………………59

Boolean conditionals:…………………………………………………………………………………………………62

Working with include:…………………………………………………………………………………………………62.

Working with handlers:………………………………………………………………………………………………63

Working with roles:………………………………………………………………………………………………………65.

Create a webserver Role:…………………………………………………………………………………………67.

Ansible vault:………………………………………………………………………………………………………………72

# Ansible

**What Is Configuration Management?**

Without automation, building and maintaining large-scale modern IT systems can be a resource-intensive undertaking and can lead to increased risk due to manual error. Configuration  management is an automated method for maintaining computer systems and software in a known, consistent state.

**How Configuration Management Works?**

There are several components in a configuration management system. Managed systems can include servers, storage, networking, and software. These are the targets of the configuration management system. The goal is to maintain these systems in known, determined states. Another aspect of a configuration management system is the description of the desired state for the systems. The third major aspect of a configuration management system is automation software, which is responsible for making sure that the target systems and software are maintained in the desired state.

**Configuration Management Benefits**:

The primary benefit of configuration management is consistency of systems and software. With configuration management, you no longer guess or hope that a configuration is current. It is correct because the configuration management system ensures that it is correct.
When combined with automation, configuration management can improve efficiency because manual configuration processes are replaced with automated processes. This also makes it possible to manage more targets with the same or even fewer resources.

# Ansible

**Why Configuration Management Is Important**

Configuration management is important because it enables the ability to scale infrastructure and software systems without having to correspondingly scale administrative staff to manage those systems. This can make it possible to scale where it previously wasn't feasible to do so.

**Configuration Management Tools**

It is common for configuration management tools to include automation too. Popular tools are:
Ansible
Chef
Puppet
Salt
Etc…

**Introduction to Ansible:**

Ansible is a general purpose automation tool that may be used for configuration management or workflow automation. Configuration management is an "infrastructure as code" practice that codifies things, e.g. what packages and versions should be installed on a system, or what daemons should be running. Workflow automation may be anything from provisioning cloud infrastructure to deploying software.

# Ansible

**IT automation:**

IT automation is the use of instructions to create a repeated process that replaces an IT professional's manual work in data centers and cloud deployments. Software tools, frameworks and appliances conduct the tasks with minimum administrator intervention. The scope of IT automation ranges from single actions to discrete sequences and, ultimately, to an autonomous IT deployment that takes actions based on user behavior and other event triggers.

**Advantages of IT automation:**

People often wonder if IT automation really brings enough advantages considering that implementing it has some direct and indirect costs. The main advantages of IT automation are:

- Ability to provision machines quickly
- Ability to recreate a machine from scratch in minutes
- Ability  to track any change performed on the infrastructure

**Types of IT automation:**

There are a lot of ways to classify IT automation systems, but by far the most important is related to how the configurations are propagated. Based on this, we can distinguish between agent-based systems and agent-less systems.

**Agent-based systems: PULL model**

Agent-based systems have two different components: a server and a client called agent.
There is only one server and it contains all of the configuration for your whole environment, while the agents are as many as the machines in the environment.Periodically, client will contact the server to see if a new configuration for its machine is present. If a new configuration is present, the client will download it and apply it.

**Agent-less systems: PUSH model**

In agent-less systems, no specific agent is present. Agent-less systems do not always respect the server/client paradigm, since it's possible to have multiple servers and even the same number of servers and clients . Communications are initialized by the server that will contact the client(s) using standard protocols (usually via SSH and PowerShell).

**What is Ansible?**

Ansible is an open-source IT automation engine which can remove drudgery from your work life, and will also dramatically improve the scalability, consistency, and reliability of your IT environment.
Ansible is a simple IT automation tool that makes your applications and  systems easier to deploy

**What Ansible can it automate?**

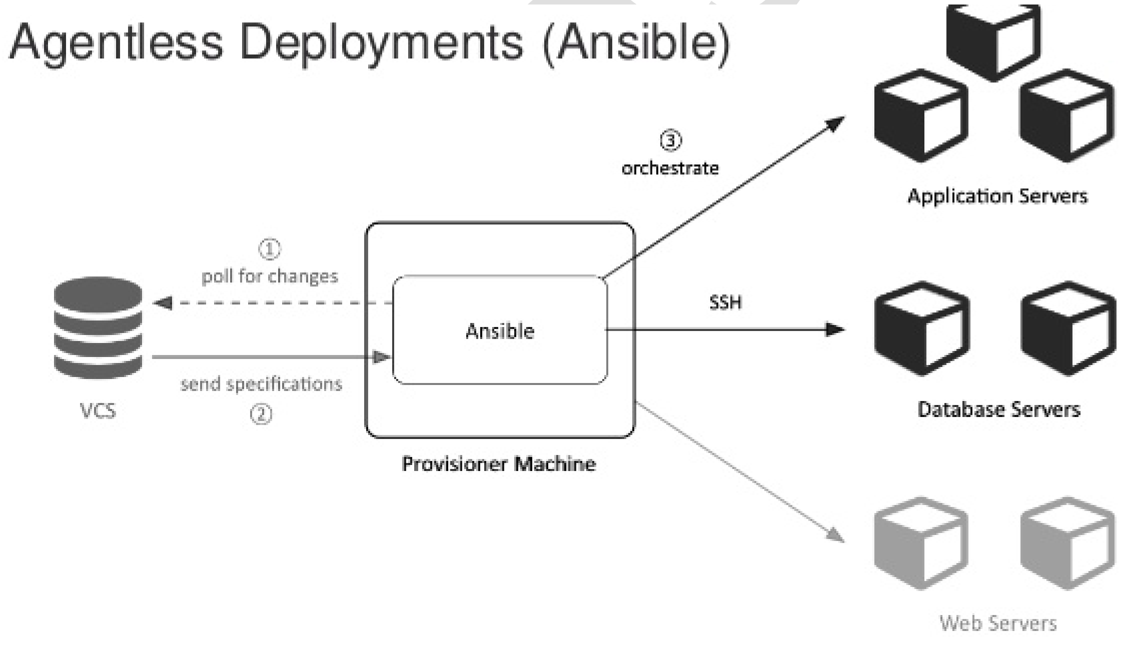
**Provisioning**: Set up the various servers you need in your infrastructure.
Configuration management: Change the configuration of an application, OS, or device; start and stop services; install or update applications; implement a security policy; or perform a wide variety of other configuration tasks.
**Application deployment**: Make DevOps easier by automating the deployment of internally developed applications to your production systems.
Ansible can automate IT environments whether they are hosted on traditional bare metal servers, virtualization platforms, or in the cloud. It can also automate the configuration of a wide range of systems and devices such as databases, storage devices, networks, firewalls, and many others.

The best part is that you don't even need to know the commands used to accomplish a particular task. You just need to specify what state you want the system to be in and Ansible will take care of it. For example, to ensure that your web servers are running the latest version of Apache, you could use a playbook similar to the following and Ansible would handle the details.

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
  - restart apache
  - name: ensure apache is running (and enable it at boot)
    service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

**How ansible works?**

Ansible works by connecting to your nodes and pushing out small programs, called "Ansible modules" to them. Ansible then executes these modules (over SSH by default), and removes them when finished. Your library of modules can reside on any machine, and there are no servers, daemons, or databases required



The management node in the above picture is the controlling node (managing node) which controls the entire execution of the playbook. It’s the node from which you are running the installation. The inventory file provides the list of hosts where the Ansible modules Ansible 3 needs to be run and the management node does a SSH connection and executes the small modules on the hosts machine and installs the product/software.

**Installation of ansible:**

**Install ansible on a Debian/Ubuntu Linux based system:**

Execute the following command on ansible master :

```
$ sudo apt-get install software-properties-common
$ sudo apt-add-repository ppa:ansible/ansible
$ sudo apt-get update
$ sudo apt-get install ansible
```

**Check ansible version command**

It is assumed that you have Ansible installed on your system. There are many documents out there that cover installing Ansible in a way that is appropriate for the operating system and version that you might be using. This book will assume the use of the Ansible 2.2.x.x version.

To discover the version in use on a system with Ansible already installed, make use of the version argument, that is, either ansible or ansible-playbook

```
$ ansible –version

  ansible 2.4.0.0
  config file = /etc/ansible/ansible.cfg
  configured module search path =
[u'/home/vivek/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/dist-
packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.14 (default, Sep 23 2017, 22:06:14) [GCC 7.2.0]
```

Note that ansible is the executable for doing adhoc one-task executions and ansible-playbook is the executable that will process playbooks for orchestrating many tasks.
The configuration for Ansible can exist in a few different locations, where the first file found will be used. The search order changed slightly in version 1.5, with the new order being:

- ANSIBLE_CFG: This is an environment variable
  ~/ansible.cfg: This is in the current directory
- ansible.cfg: This is in the user's home directory
- /etc/ansible/ansible.cfg

Inventory parsing and data sources
In Ansible, nothing happens without an inventory. Even ad hoc actions performed on localhost require an inventory, even if that inventory consists just of the localhost. The inventory is the most basic building block of Ansible architecture. When

executing ansible or ansible-playbook, an inventory must be referenced. Inventories are either files or directories that exist on the same system that runs ansible or ansible-playbook. The location of the inventory can be referenced at runtime with the --inventory-file (-i) argument, or by defining the path in an Ansible config file.

Inventories can be static or dynamic, or even a combination of both, and Ansible is not limited to a single inventory. The standard practice is to split inventories across logical boundaries, such as staging and production, allowing an engineer to run a set of plays against their staging environment for validation, and then follow with the same exact plays run against the production inventory set.

**Inventorys:**

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.
Not only is this inventory configurable, but you can also use multiple inventory files at the same time and pull inventory from dynamic or cloud sources or different formats (YAML, ini, etc), as described in Working With Dynamic Inventory. Introduced in version 2.4, Ansible has inventory plugins to make this flexible and customizable.

**Hosts and Groups:**

The inventory file can be in one of many formats, depending on the inventory plugins you have. For this example, the format for /etc/ansible/hosts is an INI-like (one of Ansible's defaults) and looks like this:

**Static inventory**

The static inventory is the most basic of all the inventory options. Typically, a static inventory will consist of a single file in the ini format. Here is an example of a static inventory file describing a single host, mastery.example.name:
That is all there is to it. Simply list the names of the systems in your inventory. Of course, this does not take full advantage of all that an inventory has to offer. If every name were listed like this, all plays would have to reference specific hostname, or the special all group. This can be quite tedious when developing a playbook that operates across different sets of your infrastructure. At the very least, hosts should be arranged into groups. A design pattern that works well is to arrange your systems into groups based on expected functionality. At first, this may seem difficult if you have an environment where single systems can play many different roles, but that is perfectly fine. Systems in an inventory can exist in more than one group, and groups can even consist of other groups! Additionally, when listing groups and hosts, it's possible to list hosts without a group. These would have to be listed first, before any other

```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
Three.example.com
```

**Ansible inventory variable data:**

Inventories provide more than just system names and groupings. Data about the systems can be passed along as well. This can include:
Host-specific data to use in templates
Group-specific data to use in task arguments or conditionals
Behavioral parameters to tune how Ansible interacts with a system
Let's improve upon our existing example inventory and add to it some variable data. We will add some host-specific data as well as group-specific data:

**Host Variables:**

```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=90
```

As alluded to above, it is easy to assign variables to hosts that will be used later in playbooks

**Group Variables:**

```
[atlanta]
host1
host2
```

```
[atlanta:vars]
ntp_server=ntp.atlanta.example.com
proxy=proxy.atlanta.example.com
```

Variables can also be applied to an entire group at once

In this example, we defined ansible_host for mastery.example.name to be the IP address of 192.168.10.25. The ansible_host variable is a behavioral inventory variable, which is intended to alter the way Ansible behaves when operating with this host. In this case, the variable instructs Ansible to connect to the system using the provided IP address rather than performing a DNS lookup on the name mastery.example.name. There are a number of other behavioral inventory variables, which are listed at the end of this section along with their intended use.

As of version 2.0, the longer form of some behavioral inventory parameters has been deprecated. The ssh part of ansible_ssh_host, ansible_ssh_user, and ansible_ssh_port is no longer required. A future release may ignore the longer form of these variables.

Our new inventory data also provides group-level variables for the web and backend groups. The web group defines http_port, which may be used in an nginx configuration file, and proxy_timeout, which might be used to determine HAProxy behavior. The backend group makes use of another behavioral inventory parameter to instruct Ansible to connect to the hosts in this group using port 314 for SSH, rather than the default of 22. Finally, a construct is introduced that provides variable data across all the hosts in the inventory by utilizing a built-in allgroup. Variables defined within this group will apply to every host in the inventory. In this particular example, we instruct Ansible to log in as the otto user when connecting to the systems. This is also a behavioral change, as the Ansible default behavior is to log in as a user with the same name as the user executing ansible or ansible-playbook on the control host.

**Ansible dynamic inventory:**

One way to setup an ec2 external inventory script is to copy the script to /etc/Ansible/ec2.py and chmod +x it. You will also need to copy the ec2.ini file to /etc/Ansible/ec2.ini. Once done, you can run Ansible as you would normally do.
For making a successful API call to AWS, you will need to configure Boto (the Python interface to AWS). There are a variety of methods available, but the simplest is to export the following environment variables:
export AWS_ACCESS_KEY_ID='AK123'
export AWS_SECRET_ACCESS_KEY='abc123'
Now, you will need to set up a few more environment variables to the inventory management script such as
$ export ANSIBLE_HOSTS=/etc/ansible/ec2.py This variable tells Ansible to use the dynamic EC2 script instead of a static /etc/ansible/hosts file. Open up ec2.py

in a text editor and make sure that the path to ec2.ini config file is defined correctly at the top of the script:

$ export EC2_INI_PATH=/etc/ansible/ec2.ini This variable tells ec2.py where the ec2.ini config file is located.

You can test the script to make sure your config is correct:

cd /etc/ansible ./ec2.py –list

After some time, you should be able to see the entire EC2 inventory across all regions in JSON. You can have a look at the scripts of EC2.py and EC2.ini

**Ad-hoc command:**

An ad-hoc command is something that you might type in to do something really quick, but don't want to save for later.

This is a good place to start to understand the basics of what Ansible can do prior to learning the playbooks language – ad-hoc commands can also be used to do quick things that you might not necessarily want to write a full playbook for.

Generally speaking, the true power of Ansible lies in playbooks. Why would you use ad-hoc tasks versus playbooks?

For instance, if you wanted to power off all of your lab for Christmas vacation, you could execute a quick one-liner in Ansible without writing a playbook.

For configuration management and deployments, though, you'll want to pick up on using '/usr/bin/ansible-playbook' – the concepts you will learn here will port over directly to the playbook language.

Example:

```
$ ansible all -i myhost  -m ping
$ ansible  all -i myhost -b  -m apt -a “name=tree”
```

**YAML:**

stands for  Yet Another Markup Language

YAML is a human-readable data serialization language. It is commonly used for configuration files, but could be used in many applications where data is being stored or transmitted.

It was designed to be easy to map  to high level  languages

Example:

If we want to list our favorite movies , student details in way  that Yaml processing engines would be able recognize.

Yaml file contains something like below.

```
---
# A list of tasty fruits
fruits:
    - Apple
    - Orange
```

- Strawberry
- Mango

**Ansible Playbooks:**

Playbooks can have a list of remote hosts, user variables, tasks, handlers, and so on. You can also override most of the configuration settings through a playbook. Let's start looking at the anatomy of a playbook.

The purpose of the playbook we are going to consider now is to ensure that the httpd package is installed and the service is enabled and started. This is the content of the apache.yaml file:

```
---
- hosts: apache
  sudo: yes
  tasks:
    - name: install apache2
      apt: name=apache2 update_cache=yes state=lates
```

**hosts**: This lists the host or host group against which we want to run the task. The hosts field is mandatory and every playbook should have it. It tells Ansible on which hosts to run the listed tasks. When provided with a host group, Ansible will take the host group from the playbook and try look for it in an inventory file . If there is no match, Ansible will skip all the tasks for that host group. The --list-hosts option along with the playbook (ansible-playbook <playbook> --list-hosts) will tell you exactly which hosts the playbook will run.

**tasks:** Finally, we come to tasks. All playbooks should contain tasks. Tasks are a list of actions you want to perform. A tasks field contains the name of the task (that is, the help text for the user about the task), a module that should be executed, and arguments that are required for

Run the playbook:

```
$ ansible-playbook -i host  apache.yml
```

The output should look like this.

```
PLAY [apache]
*****************************************************************

GATHERING FACTS
***************************************************************
ok: [111.111.111.111]

TASK: [install apache2]
*******************************************************
changed: [111.111.111.111]

PLAY RECAP
********************************************************************
111.111.111.111            : ok=2    changed=1    unreachable=0
failed=0
```

The end state, according to the playbook, has been achieved. Let's briefly look at exactly what happens during the playbook run:

```
PLAY [apache]
******************************************************************
```

This line advises us that a playbook is going to start here and that it will be executed on "all" hosts

```
GATHERING FACTS
****************************************************************
ok: [111.111.111.111]
```

Gathering info about all nodes

```
TASK: [install apache2]
********************************************************
changed: [111.111.111.111]
```

These task's states are yellow and spell "changed". This means that those tasks were executed and have succeeded but have actually changed something on the machine:

```
PLAY RECAP
*********************************************************************
111.111.111.111            : ok=2    changed=1    unreachable=0
failed=0
```

Those last few lines are a recapitulation of how the playbook went. Let's rerun the task now and see the output after both the tasks have actually run:

**Ansible verbosity:**

One of the first options anyone picks up is the debug option. To understand what is happening when you run the playbook, you can run it with the verbose (-v) option. Every extra v will provide the end user with more debug output.

Let's see an example of using the playbook debug for a single task using the following debug options with examples:

The -v option provides the default output,

```
$ ansible-playbook  -i host -v apache.yml
```

The -vv option adds a little more information:

```
$ ansible-playbook  -i host -vv apache.yml
```

The -vvv option adds a lot more information,.

```
$ ansible-playbook  -i host -vvv apache.yml
```

**Gather facts:**

"setup" module is responsible to gather facts of the remote hosts. The system facts are nothing but the system configuration which includes the hostname, IP address, filesystems, OS releases, Users, Network parameters, CPU, memory and many more. This module is automatically included in all the playbooks to gather useful variables which can be used to create the dynamic inventory or perform the specific tasks. There is a way to write the custom facts about the hosts to filter further.

```
$ ansible all -i myhosts -m  setup
```

We can obviously do the same with a playbook by gather_facts: yes, but this way is faster. Also, for the "setup" case, you will need to see the output only during the development to be sure to use the right variable name for your goal.

The output will be something like this:

```
172.31.28.158 | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "172.31.28.158"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::1b:ffff:fe6a:1d2e"
        ],
        "ansible_apparmor": {
            "status": "enabled"
        },
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "08/24/2006",
        "ansible_bios_version": "4.2.amazon",
        "ansible_cmdline": {
            "BOOT_IMAGE": "/boot/vmlinuz-
3.13.0-158-generic",
            "console": "ttyS0",
            "ro": true,
            "root": "UUID=1f08415c-7898-4e3a-
b006-902dd2bebf5f"
        },
        "ansible_date_time": {
            "date": "2019-01-29",
            "day": "29",
            "epoch": "1548761266",
            "hour": "11",
            "iso8601": "2019-01-29T11:27:46Z",
            "iso8601_basic":
"20190129T112746308470",
            "iso8601_basic_short":
"20190129T112746",
            "iso8601_micro": "2019-01-
29T11:27:46.308538Z",
            "minute": "27",
            "month": "01",
            "second": "46",
            "time": "11:27:46",
            "tz": "UTC",
            "tz_offset": "+0000",
            "weekday": "Tuesday",
            "weekday_number": "2",
            "weeknumber": "04",
            "year": "2019"
        },
        "ansible_default_ipv4": {
            "address": "172.31.28.158",
            "alias": "eth0",
            "broadcast": "172.31.31.255",
            "gateway": "172.31.16.1",
            "interface": "eth0",
            "macaddress": "02:1b:ff:6a:1d:2e",
            "mtu": 9001,
            "netmask": "255.255.240.0",
            "network": "172.31.16.0",
            "type": "ether"
        },
        "ansible_default_ipv6": {},
        "ansible_device_links": {
            "ids": {},
            "labels": {
                "xvda1": [
                    "cloudimg-rootfs"
                ]


        "ansible_distribution": "Ubuntu",
        "ansible_distribution_file_parsed":
true,
        "ansible_distribution_file_path":
"/etc/os-release",
        "ansible_distribution_file_variety":
"Debian",
        "ansible_distribution_major_version":
"14",
        "ansible_distribution_release":
"trusty",
        "ansible_distribution_version":
"14.04",
        "ansible_dns": {
```

```
            "nameservers": [
                "172.31.0.2"
            ],
            "search": [
                "ap-southeast-
1.compute.internal"
            ]
        },
        "ansible_domain": "ap-southeast-
1.compute.internal",
        "ansible_effective_group_id": 1001,
        "ansible_effective_user_id": 1001,
        "ansible_env": {
            "HOME": "/home/maha",
            "LANG": "en_US.UTF-8",
            "LOGNAME": "maha",
            "MAIL": "/var/mail/maha",
            "PATH":
"/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/bin:/usr/games:/usr/local/games",
            "PWD": "/home/maha",
            "SHELL": "/bin/bash",
            "SHLVL": "1",
            "SSH_CLIENT": "172.31.28.158 48815
22",
            "SSH_CONNECTION": "172.31.28.158
48815 172.31.28.158 22",
            "SSH_TTY": "/dev/pts/2",
            "TERM": "xterm",
            "USER": "maha",
            "XDG_RUNTIME_DIR":
"/run/user/1001",
            "XDG_SESSION_ID": "2",
            "_": "/bin/sh"
        },
        "ansible_eth0": {
            "active": true,
            "device": "eth0",
            "features": {

            "hw_timestamp_filters": [],
            "ipv4": {
                "address": "172.31.28.158",
                "broadcast": "172.31.31.255",
                "netmask": "255.255.240.0",
                "network": "172.31.16.0"
            },
            "ipv6": [
                {
                    "address":
"fe80::1b:ffff:fe6a:1d2e",
                    "prefix": "64",
                    "scope": "link"
                }
            ],
            "macaddress": "02:1b:ff:6a:1d:2e",
                "mtu": 9001,
                "pciid": "vif-0",
                "promisc": false,
                "timestamping": [
                    "rx_software",
                    "software"
                ],
                "type": "ether"
            },
            "ansible_fips": false,
            "ansible_form_factor": "Other",
            "ansible_fqdn": "ip-172-31-28-158.ap-
southeast-1.compute.internal",
            "ansible_hostname": "ip-172-31-28-158",
            "ansible_interfaces": [
                "lo",
                "eth0"
            ],
            "ansible_is_chroot": false,
            "ansible_iscsi_iqn": "",
            "ansible_kernel": "3.13.0-158-generic",
            "ansible_lo": {
                "active": true,
                "device": "lo",
                "features": {
                    "fcoe_mtu": "off [fixed]",
                    "generic_receive_offload":
"on",
                    "generic_segmentation_offload":
    "on",
                    "highdma": "on [fixed]",
                    "l2_fwd_offload": "off
    [fixed]",
                    "large_receive_offload": "off
    [fixed]",
                    "loopback": "on [fixed]",
                    "netns_local": "on [fixed]",
                    "ntuple_filters": "off
    [fixed]",

                "hw_timestamp_filters": [],
                "ipv4": {
                    "address": "127.0.0.1",
                    "broadcast": "host",
                    "netmask": "255.0.0.0",
                    "network": "127.0.0.0"
                },
                "ipv6": [
                    {
                        "address": "::1",
                        "prefix": "128",
                        "scope": "host"
                    }
                ],
                "mtu": 65536,
                "promisc": false,
                "timestamping": [
                    "rx_software",
                    "software"
                ],
                "type": "loopback"
            },
            "ansible_local": {},
            "ansible_lsb": {
                "codename": "trusty",
                "description": "Ubuntu 14.04.5
    LTS",
                "id": "Ubuntu",
                "major_release": "14",
                "release": "14.04"
            },
            "ansible_machine": "x86_64",
            "ansible_machine_id":
    "341d3ce0126bb8d72ab300a05c4eb61f",
            "ansible_memfree_mb": 782,
            "ansible_memory_mb": {
                "nocache": {
                    "free": 880,
                    "used": 112
                },
                "real": {
                    "free": 782,
                    "total": 992,
                    "used": 210
                },
                "swap": {
                    "cached": 0,
                    "free": 0,
                    "total": 0,
                    "used": 0
                }
            },
            "ansible_memtotal_mb": 992,
            "ansible_mounts": [
                {
            "ansible_selinux_python_present":
    false,
            "ansible_service_mgr": "upstart",
            "ansible_ssh_host_key_dsa_public":
    "AAAAB3NzaC1kc3MAAACBAPAuku5YRe4wJaiXR3Ogt71gYl
    z+saWRx//2+igaroBbqXXVSY3hmbY54p8QexpD+ojFcOjbQ
    +rUorenqh9/A7MiWD57ptcVcnWMYN9nCa43ulEI/FXSVWUN
    C/wu2kn4PTYI87auascYdL2/vkImugKIIkjsfNY/MXAcznt
    pr8gjAAAAFQDRQLGSwEIw9CERnhoeQ1pkuyI77wAAAIEAvq
    hP7UqJIYUMWECVewK6EgPrL26cA2JiPYuOGYF2Koh9qy2DU
    yUqt5MSX+4HpEvAZCWoxuxsgaEccZjLrgYwt2FS60xJkdLh
    lpO6gF9/cTAQedYWW/4C4sqgV6s69hiGs1253lkUwa8P2F/
    Kg1Go8qI/7NrFFEZmImrUojN4v1kAAACBAMccTVcukRbDF6
    Ar27r3dcFgq6QEfD6rBnrgImCTg1oa6W2n0Jvo+Un8UvxqR
    oXDO/U1fWPrxQcM80UaSvJ8Ev3x9BB8Ag6mss8br3c0s5wM
    0TR0RNNiGExCOuicTTF1ynD1nl+Sqyr/ZNPCSvnUWXBAN2B
    mC3hRBzvzw78S9pQS",
            "ansible_ssh_host_key_ecdsa_public":
    "AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNT
    YAAABBBBuEKTjGhF1fAJ13sXx00PsMVOuuvfuDCTN3TZj6u
    LL+0PXxZrZ7Pwd46qe7U8++A0DOIJS+KRLn0qWMYndieRw=
    ",
            "ansible_ssh_host_key_ed25519_public":
    "AAAAC3NzaC1lZDI1NTE5AAAAIEO2AMbJAE5vdiiNPZ1mjQ
    VyzH0CWwVvnVTmFAnihAI6",
            "ansible_ssh_host_key_rsa_public":
    "AAAAB3NzaC1yc2EAAAADAQABAAABAQCu9ILaSH4jIq5p1q
    sDiiqaKA+91ziWFwhlq9Nt5+1ziUCEjCONLU6ylXp/7MQqW
    pqToOhEMwlosVMx3bC3su2eLQ0zW+w1uWNpK4toNmcqKGnX
    8Xj6aiZC6ivEXssJL1Ts0F47Jt1bI2hjlztSlQ9pOFyZ613
    n0aGnEyCBsy7wRzne1S9WiF0SZOBeyTB64DiEImEEVYEOZi
    HTz8cpfMnKme7U68kZt+kOYCHo2zJVQkgsnXfdFWaeNCew2
    B0PkmFVYCHz4NPCJxwBkV9udeECyxPnim4czwhJE9v4N0B1
    a2zzyIiTAVVVdTF9Mt6kHfb0hIcwhZRqxGaQJ+HRMvaJ",
            "ansible_swapfree_mb": 0,
            "ansible_swaptotal_mb": 0,
            "ansible_system": "Linux",
            "ansible_system_capabilities": [
                ""
            ],
            "ansible_system_capabilities_enforced":
    "True",
            "ansible_system_vendor": "Xen",
            "ansible_uptime_seconds": 2309,
            "ansible_user_dir": "/home/maha",
            "ansible_user_gecos": ",,,",
            "ansible_user_gid": 1001,
            "ansible_user_id": "maha",
            "ansible_user_shell": "/bin/bash",
            "ansible_user_uid": 1001,
            "ansible_userspace_architecture":
    "x86_64",
            "ansible_userspace_bits": "64",
            "ansible_virtualization_role": "guest",
            "ansible_virtualization_type": "xen",
            "gather_subset": [
                "all"
            ],
            "module_setup": true
        },
        "changed": false
    }
```

As you can see, from this huge list of options, you can gain a huge quantity of information, and you can use them as any other variable. Let's print the OS name and the version. To do so, we can create a new playbook called debugwithvars.yamlwith the following content:

```
---
- hosts: all
become: yes
tasks:
- name: display the os
```

```
debug:
msg:"{{ansible_distribution}}"
- name: display version
debug:
msg:"{{ansible_distribution_version}}"
```

Run as below:

```
$ ansible-playbook -i myhosts debugwithvars.yml
```

This will give us the following output:

```
PLAY [all]
*************************************************************
********************************

TASK [Gathering Facts]
*************************************************************
********************
ok: [172.31.28.158]

TASK [display the os]
*************************************************************
*********************
ok: [172.31.28.158] => {
    "msg": "Ubuntu"
}

TASK [display version]
*************************************************************
********************
ok: [172.31.28.158] => {
    "msg": "14.04"
}

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

We create the variables in playbook and we can only use in same playbooks as below mention.

```
---
- hosts: all
become: yes
vars:
var1: git
var2: tree
var3: wget
tasks:
- name: display the var2
debug:
msg:"{{var2}}"
```

```
$ ansible-playbook -i myhosts debugwithvars.yml
```

It will be followed output:

```
PLAY [all]
************************************************************
********************************

TASK [Gathering Facts]
************************************************************
********************
ok: [172.31.28.158]

TASK [display the var2]
************************************************************
*******************
ok: [172.31.28.158] => {
    "msg": "tree"
}

PLAY RECAP
************************************************************
********************************
172.31.28.158              : ok=2    changed=0
unreachable=0    failed=0
```

**Upgrading all installed packages:**

To upgrade all installed packages, we will need to use the apt module again, but with a different parameter, in fact we would use:

```
---
- hosts: all
become: yes
tasks:
- name: Upgrading all installed packages
apt:
name:"*"
state: latest
```

As you can see, we have specified "*" as the package name (this stands for a wildcard to match all installed packages) and the state is latest. This will upgrade all installed packages to the latest version available.

If you remember, when we talked about the "present" state, we said that it was going to install the last available version. So what's the difference between "present" and "latest"? Present will install the latest version if the package is not installed, while if the package is already installed (no matter the version) it will go forward without making any change. Latest will install the latest version if the package is not installed, while if the package is already installed will check whether a newer version is available and if it is, Ansible will update the package.

**Creating the Ansible user:**

When you create a machine (or rent one from any hosting company) it arrives only with the root user. Let's start creating a playbook that ensures that an Ansible user is created, it's accessible with an SSH key, and is able to perform actions on behalf of other users (sudo) with no password asked. I often call this playbook, create.yaml since I execute it as soon as a new machine is created, but after that, I don't use it since it uses the root user that I disable for security reasons. Our script will look something like the following:

```
---
- hosts: all
become: yes
tasks:
- name: creating user
user:
name: ansible
state: present
comment: ansible

$ ansible-playbook -i myhosts creatuser.yml
```

It will be gives follow output:

```
PLAY [all]
**********************************************************************
**********************

TASK [Gathering Facts]
**********************************************************************
**********
ok: [172.31.28.158]

TASK [creating user]
**********************************************************************
************
ok: [172.31.28.158]

PLAY RECAP
**********************************************************************
**********************
172.31.28.158              : ok=2    changed=0    unreachable=0
failed=0
```

**Install FTP server:**

We can install and start ftp server by using as below playbook.

```
---
- hosts: all
become: yes
tasks:
- name: install  ftp server
apt:
name: vsftpd
state: present

- name: start ftp server
service:
name: vsftpd
state: started
enabled: yes

$ ansible-playbook -i myhosts ftpinstall.yml
```

It will be gives follow output:

```
PLAY [all]
************************************************************
********************************
```

```
TASK [Gathering Facts] ***********************************************************************************
ok: [172.31.28.158]

TASK [install  ftp server] *******************************************************************************
ok: [172.31.28.158]

TASK [start ftp server] **********************************************************************************
ok: [172.31.28.158]

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

Same as above we can install  and configure any server like ( NFS, DNS,NTP, DHCP, and Network .....etc).

**Install webserver**:

Now that we have made some generic changes to the operating system, let's move on to actually creating a web server. We are splitting those two phases so we can share the first phase between every machine and apply the second only to web servers.

For this second phase, we will create a new playbook called webserver.yml with the following content:

```
---
- hosts: all
become: yes
tasks:
- name: install webserver on ubuntu(apache2)
apt:
name: apache2
state: present
- name: start webserver on ubuntu(apache2)
service:
name: apache2
state: started
```

**note**:if you want install webserver on ubuntu server we can use apt modile and apache2 package, if you want to install webserver on redhat server we can use yum module and httpd package

Run above playbook by using as follows:

```
$ ansible-playbook -i myhosts webserver.yml
```

It will be gives following output

```
PLAY [all]
************************************************************
********************************
```

```
TASK [Gathering Facts]
****************************************************************************
ok: [172.31.28.158]

TASK [install webserver on ubuntu(apache2)]
***********************************************************
changed: [172.31.28.158]

TASK [start webserver on ubuntu(apache2)]
************************************************************
*
ok: [172.31.28.158]

PLAY RECAP
************************************************************************************************
172.31.28.158              : ok=3    changed=1
unreachable=0    failed=0
```

**Jinja2 templates:**

Jinja2 is a widely-used and fully-featured template engine for Python. Let's look at some syntax that will help us with Ansible. This paragraph does not want to be a replacement for the official documentation, but its goal is to teach you some components that you'll find very useful when using them with Ansible.
As we have seen, we can print variable content simply with the'{{ VARIABLE_NAME }}' syntax. If we want to print just an element of an array we can use '{{ ARRAY_NAME['KEY']}}', and if we want to print a property of an object, we can use '{{OBJECT_NAME.PROPERTY_NAME }}'.

Our website will be a simple, single page website, we can easily create it and publish it using a single Ansible task. To make this page a little bit more interesting, we will create it from a template that will be populated by Ansible with a little data about the machine. The script to publish it will be called webserver_template.yml and will have the following content:

```
---
- hosts: all
become: yes
tasks:
- name: install apache2 on webserver
apt:
name: apache2
state: present

- name: copy file(index.html) into webserver
template:
src: myindex.j2
dest: /var/www/html/index.html

- name: start webserver
service:
name: apache2
state: started
```

Let's start with a simple template that we will call myindex.j2:

```
<html>
<body>
```

```
<h1>My First Heading</h1>
<p>My first paragraph date on {{ansible_date_time.date}}</p>
</body>
</html>
```

We can deploy website as running below

```
$ ansible-playbook -i myhosts webserver_template.yml
```

It will be followed output:

```
PLAY [all]
*************************************************************
********************************

TASK [Gathering Facts]
*************************************************************
********************
ok: [172.31.28.158]

TASK [install apache2 on webserver]
*************************************************************
*******
ok: [172.31.28.158]

TASK [copy file(index.html) into webserver]
************************************************************
changed: [172.31.28.158]

TASK [start webserver]
*************************************************************
********************
ok: [172.31.28.158]
```

```
PLAY RECAP
*************************************************************
********************************
172.31.28.158
```

**Jinja2 templates Conditionals controls:**

Ansible uses Jinja2 as a template engine. Hence, it would be useful for us to understand Jinja2 control structures in addition to the ones supported by Ansible tasks. Jinja2's syntax encloses the control structures inside the {% %}blocks. For conditional control, Jinja2 uses the familiar if statements, which have the following syntax:

Syntex:

```
{% if condition %}
    do_some_thing
{% elif condition2 %}
    do_another_thing
{% else %}
    do_something_else
{% endif %}
```

Example:

```
<html>
<body>
```

```
<h1>My First Heading</h1>
<p>My first paragraph date on {{ansible_date_time.date}}</p>
{%if ansible_eth0.active is equalto True%}
<p>{{ansible_all_ipv4_addresses}}</p>
{% endif %}
</body>
</html>
```

As you can see, we have added the capability to print the main IPv4 address for the eth0 connection, if the connection is active. With conditionals we can also use the tests.

**Working with inventory files:**

An inventory file is the source of truth for Ansible. It follows the INI format and tells Ansible whether the remote host or hosts provided by the user are genuine or not.
Ansible can run its tasks against multiple hosts in parallel. To do this, you can directly pass the list of hosts to Ansible using an inventory file. For such parallel execution, Ansible allows you to group your hosts in the inventory file; the file passes the group name to Ansible. Ansible will search for that group in the inventory file and run its tasks against all the hosts listed in that group.
You can pass the inventory file to Ansible using the -i or --inventory-file option followed by the path to the file. If you do not explicitly specify any inventory file to Ansible, it will take the default path from the host_fileparameter of ansible.cfg, which defaults to /etc/ansible/hosts.

**The basic inventory file:**

Before diving into the concept, first let's look at a basic inventory file in the following  myhost file

```
Example.com
172.31.28.158
172.168.1.48
172.168.2.38
ip-172-31-28-158.ap-southeast-1.compute.internal
```

Ansible can take either a hostname or an IP address within the inventory file. In the preceding example, we specified four servers; Ansible will take these servers and search for the hostname that you provided, to run its tasks. If you want to run your Ansible tasks against all of these hosts, then you can pass "**all**" to the hosts parameter while running the ansible-playbook or to the ansible command; this will make Ansible run its tasks against all the hosts listed in an inventory file.

You can use ansible command as below

```
$  ansible all –i myhost –m ping
```

You can use ansible-playbook command as below

```
$ ansible-playbook –i myhost  paybookname.yml
---
- hosts: all
tasks:
```

**Groups in an inventory file:**

In the following example, we grouped the inventory file into three groups, that is, webservers, applicationservers, and DBservers

```
[webservers]
172.31.28.158
172.31.80.45

[applicationservers]
172.31.21.117
172.31.80.45

[DBservers]
172.31.80.45
172.31.87.65
```

Now, instead of running Ansible against all the hosts, you can run it against a set of hosts by passing the group name to the ansible-playbook command. When Ansible runs its tasks against a group, it will take all the hosts that fall under that group. To run Ansible against the "applicationserver" group, you need to run the commands line shown in the following.

You can use ansible command as below

```
$  ansible applicationserver –i myhost –m ping
```

You can use ansible-playbook command as below

```
$ ansible-playbook –i myhost  paybookname.yml
```

```
---
- hosts:applicationserver
tasks:
```

**Regular expressions in the inventory file:**

An inventory file would be very helpful if you have many servers. Let's say you have a large number of web servers that follow the same naming convention, for example, 172.31.87.224001, 172.31.87.224002, …, 172.31.87.22400N, and so on. Listing all these servers separately will result in a dirty inventory file, which would be difficult to manage with hundreds to thousands of lines. To deal with such situations, Ansible allows you to use regex inside its inventory file.
The following shows an example of multiple servers:

```
[webservers]
172.31.87.224[001:200]

[applicationservers]
172.31.28.158[001:020]
```

**Working with variables:**

Ansible allows you to define external variables in many ways, from an external variable file within a playbook, by passing it from the Ansible command using the -e / --extra-vars option, or by passing it to an inventory file. You can define external variables in an inventory file either on a per-host basis, for an entire group, or by creating a variable file in the directory where your inventory file exists.

**Host variables:**

It's possible to declare variables for a specific host, declaring them in the hosts file. For instance, we may want to specify different engines for our web servers. Let's suppose that one needs to reply to a specific domain, while the other to a different domain name. In this case, we would do it with the following hosts file:

Using the following inventory file, you can access the variable db_name for the 172.31.80.45host, and db_name and db_port for 172.31.87.65

```
[applicationservers]
172.31.28.158
172.31.80.45

[DBservers]
172.31.80.45db_name=oracle
172.31.87.65db_name=mysql db_port=3306
```

**Group variables:**

There are other cases where you want to set a variable that is valid for the whole group. Let's suppose that we want to declare the variable https_enabled to True and its value has to be equal for all web servers. In this case, we can create a [webserver:vars] section,

Let's move to variables that can be applied to a group. Consider the following.

```
[webservers]
```

```
172.31.87.224

[applicationservers]
172.31.28.158

[webservers:vars]
Webpack=apache2
Apache2_enabled=yes
```

**Variable files:**

Apart from host and group variables, you can also have variable files. Variable files can be either for hosts or groups that reside in the folder of your inventory file. All of the host variable files will go under the host_varsdirectory, whereas all group variable files will go under the group_vars directory.

**Overriding configuration parameters with an inventory file:**

You can override some of Ansible's configuration parameters directly through the inventory file. These configuration parameters will override all the other parameters that are set either through ansible.cfg, environment variables, or passed to the ansible-playbook/ansible command. The following is the list of parameters you can override from an inventory file:

- ansible_ssh_user: This parameter is used to override the user that will be used for communicating with the remote host.
- ansible_ssh_port: This parameter will override the default SSH port with the user-specified port. It's a general, recommended sysadmin practice to not run SSH on the standard port 22.
- ansible_ssh_host: This parameter is used to override the host for an alias.
- ansible_connection: This specifies the connection type that should be used to connect to the remote host. The values are SSH, paramiko, or local.

- ansible_ssh_private_key_file: This parameter will override the private key used for SSH; this will be useful if you want to use some specific keys for a specific host. A common use case is if you have hosts spread across multiple data centers, multiple AWS regions, or different kinds of applications. Private keys can potentially be different in such scenarios.
- ansible_shell_type: By default, Ansible uses the sh shell; you can override this using the ansible_shell_type parameter. Changing this to csh, ksh, and so on will make Ansible use the commands of that shell.
- ansible_python_interpreter: Ansible, by default, tries to look for a Python interpreter within /usr/bin/python; you can override the default Python interpreter using this parameter.

Let's take a look at the following example:

```
[applicationservers]
172.31.28.158 ansible_ssh_user=myuser ansible_ssh_private_key_file=myuser.rsa
172.31.80.45
```

**Working with dynamic inventory:**

A user using configuration management system will often want to save inventory in a different software system. As described in inventory, a basic text-based system is provided by Ansible. Some examples include pulling inventory from a cloud provide.

These options are supported by Ansible through an external inventory system. Some of these such as EC2 Cloud

Maintaining an inventory file might not be the best approach if you use Amazon Web Services EC2. This is because the hosts may vary over time, they can be managed by external applications, or you might be using AWS autoscaling. In such a case, you can use the EC2 external inventory script.

**Setting up EC2 External Inventory Script With Ansible:**

One way to setup an ec2 external inventory script is to copy the script to /etc/Ansible/ec2.py and chmod +x it. You will also need to copy the ec2.ini file to /etc/Ansible/ec2.ini. Once done, you can run Ansible as you would normally do. For making a successful API call to AWS, you will need to configure Boto (the Python interface to AWS). There are a variety of methods available, but the

simplest is to export the following environment variables:

```
export AWS_ACCESS_KEY_ID='AK123'
export AWS_SECRET_ACCESS_KEY='abc123'
```

Now, you will need to set up a few more environment variables to the inventory management script such as

$ export ANSIBLE_HOSTS=/etc/ansible/ec2.py This variable tells Ansible to use the dynamic EC2 script instead of a static /etc/ansible/hosts file. Open up ec2.py in a text editor and make sure that the path to ec2.ini config file is defined correctly at the top of the script:

$ export EC2_INI_PATH=/etc/ansible/ec2.ini This variable tells ec2.py where the ec2.ini config file is located.

You can test the script to make sure your config is correct:

```
$ ansible./ec2.py –list
$ ansible-playbook -u ./ec2.py  ping.yml
```

After some time, you should be able to see the entire EC2 inventory across all regions in JSON. You can have a look at the scripts of EC2.py and EC2.ini

**Working Ansible loops:**

You may have noticed that up to now we have never used cycles, so every time we had to do multiple, similar operations, we wrote the code multiple times. An example of this is the webserver.yaml code.

In fact, this was the content of the webserver.yaml file:

```
---
- hosts: all
become: yes
tasks:
- name:install git on all my nodes
apt:
name: git
state: present
- name:install tree on  nodes
apt:
name: tree
state: present

- name: install wget on  nodes
apt:
name: wget
state: present
```

**Standard loops - with_items:**

To improve the above code, we can use a simple iteration: with_items.
This allows us to iterate in a list of item, and at every iteration, the designated item of the list will be available to us in the item variable.
We can therefore change that code to the following

```
---
- hosts: all
become: yes
gather_facts: yes
tasks:
- name: install git, tree and wget on ubuntu
apt:
name:"{{ item }}"
state: present
with_items:
- git
- tree
- wget
```

Let's run playbook.

```
$ ansible-playbook -i myhosts iterates.yml
```

It will be followed output:

```
PLAY [all]
*************************************************************
********************************
```

```
TASK [Gathering Facts]
******************************************************************************
ok: [172.31.28.158]

TASK [install git, tree and wget on ubuntu]
************************************************************
changed: [172.31.28.158] => (item=[u'git', u'tree',
u'wget'])

PLAY RECAP
****************************************************************************************
172.31.28.158              : ok=2    changed=1
unreachable=0    failed=0
```

**Nested loops - with_nested:**

In addition to the simple loops we described previously, Ansible's syntax also supports the idea of nested looping. Nested loops in many ways are similar in nature to a set of arrays that would be iterated over using the with_nested operator. Nested loops provide us with a succinct way of iterating over multiple lists within a single task. This could be useful in cases where multiple data items are required (such as creating user accounts with different names and details, or maybe seeding a MySQL database). Let's look at an example

```
- name: give users access to multiple databases
mysql_user:
name:"{{ item[0] }}"
priv:"{{ item[1] }}.*:ALL"
append_privs: yes
password:"foo"
```

```
with_nested:
- ['alice', 'bob']
- ['clientdb', 'employeedb', 'providerdb']
```

**Delegating a task:**

Sometimes you want to execute an action on a different system. This could be, for instance, a database node while you are deploying something on an application server node or to the local host. To do so, you can just add the 'delegate_to: HOST' property to your task and it will be run on the proper node.

```
---
- hosts: all
remote_user: maha
tasks:
- name: Count processes running on the remote system
shell: ps | wc -l
register: remote_processes_number
- name: Print remote running processes
debug:
msg:'{{ remote_processes_number.stdout }}'
- name: Count processes running on the local system
shell: ps | wc -l
delegate_to: localhost
register: local_processes_number
- name: Print local running processes
debug:
msg:'{{ local_processes_number.stdout }}'
```

Let's run this  playbooks

```
$ ansible-playbook -i myhosts delegating.yml
```

It will be followed output

```
PLAY [all]
************************************************************
********************************

TASK [Gathering Facts]
************************************************************
********************
ok: [172.31.28.158]

TASK [Count processes running on the remote system]
***************************************************
changed: [172.31.28.158]

TASK [Print remote running processes]
************************************************************
*****
ok: [172.31.28.158] => {
    "msg": "6"
}

TASK [Count processes running on the local system]
****************************************************
changed: [172.31.28.158 -> localhost]

TASK [Print local running processes]
************************************************************
******
ok: [172.31.28.158] => {
    "msg": "10"
}
```

```
PLAY RECAP
*************************************************************
*********************************
172.31.28.158              : ok=5    changed=2
unreachable=0    failed=0
```

**Working with ansible conditions: when**

Basic Usage:

Use the when condition to control whether a task or role runs or is skipped. This is normally used to change play behavior based on facts from the destination system. Consider this playbook:

```
---
- hosts: all
tasks:
- include: Ubuntu.yml
when: ansible_os_family == "Ubuntu"

- include: RHEL.yml
when: ansible_os_family == "RedHat"
```

Having two tasks, one for the httpd package (for Red-Hat-based systems) and the other for the apache2 package (for Debian-based systems) in a playbook, will make Ansible install both packages, and this execution will fail, as apache2 will not be available if you're installing on a Red-Hat-based operating system. To overcome such problems, Ansible provides conditional statements that help run a task only when a specified condition is met.

While installing httpd on a Red-Hat-based operating system, we first check whether the remote system is running a Red-Hat-based operating system, and if it is, we then install the httpd package; otherwise, we skip the task. Without wasting your time, let's dive into an example playbook called webserver.yaml with the following content:

```
---
- hosts: all
become: yes
tasks:
- name: install webserver on ubuntu(apache2)
apt:
name: apache2
state: present
when: ansible_os_family =="Debian"
- name: start webserver on ubuntu(apache2)
service:
name: apache2
state: started
when: ansible_os_family =="Debian"
- name: install webserver on Redhat ( httpd )
yum:
name: httpd
state: present
when: ansible_os_family =="RedHat"
- name: start webserver on Redhat
service:
name: httpd
state: started
when: ansible_os_family =="RedHat"
```

Run it with the following:

```
$ ansible-playbook -i myhosts webserveronUR.yml
```

It will be followed output:

```
PLAY [all]
************************************************************
********************************

TASK [Gathering Facts]
************************************************************
********************
ok: [172.31.28.158]

TASK [install webserver on ubuntu(apache2)]
***********************************************************
ok: [172.31.28.158]

TASK [start webserver on ubuntu(apache2)]
************************************************************
*
ok: [172.31.28.158]

TASK [install webserver on Redhat ( httpd )]
**********************************************************
skipping: [172.31.28.158]

TASK [start webserver on Redhat]
************************************************************
**********
skipping: [172.31.28.158]
```

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

**Boolean conditionals:**

Apart from string matching, you can also check whether a variable is True. This type of validation will be useful when you want to check whether a variable was assigned a value or not. You can even execute a task based on the Boolean value of a variable.

**Working with include:**

Includes a file with a list of plays or tasks to be executed in the current playbook. To trigger the inclusion of another file, you need to put the following under the tasks object:

- include: Ubuntu.yml

In addition of passing variables, you can also use conditionals to include a file only when certain conditions are matched, for instance to include the redhat.yaml file only if the machine is running an OS in the Red Hat family using the following code:

```
---
- hosts: all
tasks:
- include: Ubuntu.yml
when: ansible_os_family == "Ubuntu"

- include: RHEL.yml
when: ansible_os_family == "RedHat"
```

**Working with 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.

The best way to think of this is using the example of having a Playbook for installing Apache. As part of that Playbook, we have a variety of Tasks that carry out the essential tasks that make up our Apache install. Everything from installing Apache through to securing our installation, creating our expected directory structure, and ensuring all our plugins are installed.

Now, as part of some of these Tasks we would have to re-start Apache.

We don't want to copy and paste a Task over and over, whenever we need to re-start Apache .

Instead, we want to put a single Restart Apache Task into our Playbook somehow, and then call that Task as required.

That Task we call is a Handler. A task we can re-use multiple times whilst our Playbook runs.We call Handlers using the notify syntax. As following palybook.

```
---
- hosts: all
become: yes
tasks:
- name: install webserver on ubuntu
yum:
name: apache2
state: present
notify: startapache2

handlers:
- name: startapache2
service:
name: apache2
state: started
```

let's run this playbook

```
$ ansible-playbook -i myhosts webserver_handlers.yml
```

It will be followed output

```
PLAY [all]
************************************************************
********************************

TASK [Gathering Facts]
************************************************************
********************
ok: [172.31.28.158]

TASK [install webserver on ubuntu]
************************************************************
********
```

```
ok: [172.31.28.158]

PLAY RECAP
************************************************************
********************************
172.31.28.158              : ok=2    changed=0
unreachable=0    failed=0
```

**Working with roles:**

Simply put, roles are a further level of abstraction that can be useful for organizing playbooks. As you add more and more functionality and flexibility to your playbooks, they can become unwieldy and difficult to maintain as a single file. Roles allow you to create very minimal playbooks that then look to a directory structure to determine the actual configuration steps they need to perform

Within Ansible there are two techniques for reusing a set of configuration management tasks, includes and roles. Although both techniques function in similar ways, roles appear to be the official way forward. Ansible Galaxywas built as a repository for roles, and as we'll see in this post, ansible-galaxy exists to aid in installing and creating them.

```
$ ansible-galaxy init myrole
```

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

These are the directories that will contain all of the code to implement our configuration. You may not use all of the directories, so in real practice, you may not need to create all of these directories.

This is what they are all for:

- files: This directory contains regular files that need to be transferred to the hosts you are configuring for this role. This may also include script files to run.
- handlers: All handlers that were in your playbook previously can now be added into this directory.
- meta: This directory can contain files that establish role dependencies. You can list roles that must be applied before the current role can work correctly.

- templates: You can place all files that use variables to substitute information during creation in this directory.
- tasks: This directory contains all of the tasks that would normally be in a playbook. These can reference files and templates contained in their respective directories without using a path.
- vars: Variables for the roles can be specified in this directory and used in your configuration files

Within all of the directories but the "files" and "templates", if a file called main.yml exists, its contents will be automatically added to the playbook that calls the role.

**Create a webserver Role:**

Let's create the webserer role by follow command

```
$ ansible-galaxy init webserverrole
```

It will be created as follow:

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

```
    └── main.yml
```

As we transformed the common playbook into the common role, we can do the same for the webserver role.
In roles, we need to have the webserver folder with the tasks subfolder inside it. In this folder, we have to put the main.yaml file containing the tasks copied from the playbooks, that should look like:

Webserverrole/tasks/main.yml

```
---
# tasks file for webserverrole
- name: install apache2 on webserver
apt:
name: apache2
state: present

- name: copy file(index.html) into webserver
template:
src: myindex.j2
dest: /var/www/html/index.html
notify: startapche2
```

Webserverrole/templates/myindex.j2

```
<html>
<body>
<h1>My First Heading</h1>
<p>My first paragraph date on {{ansible_date_time.date}}</p>
</body>
</html>
```

Webserverrole/ handlers /main.yml

```
---
# handlers file for webserverrole
- name: startapache2
service:
name: apache2
state: started
```

The only thing that we still need to do to make our role applicable is to add the entry in the webserver.yaml file so that Ansible is informed that the servers in the webserver group should apply the webserver role as well as the common role.webserver.yaml will need to be like the following:

```
---
- hosts: webserver
become: yes
roles:
- {role: webserverrole, when: ansible_os_family =="Debian"}
```

we can also just execute the webserver.yml, since the change we just did is relevant only to this group of servers. To do so we run:

```
$ ansible-playbook -i myhosts webserver.yml
```

It will be followed outout:

```
PLAY [all]
************************************************************
********************************

TASK [Gathering Facts]
************************************************************
********************
ok: [172.31.28.158]

TASK [webserverrole : install apache2 on webserver]
***************************************************
ok: [172.31.28.158]

TASK [webserverrole : copy file(index.html) into webserver]
*******************************************
ok: [172.31.28.158]

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

As you can see, the webserver roles has been applied to the webserver nodes. It's very important to apply all roles concerning a specific node and not just the one you changed because more often than not, when there is a problem on one or more nodes in a group but not on other nodes of the same group, the problem is some roles have been applied unequally in the group. Only by applying all concerned roles to a group, will it grant you the equality of the nodes of that group.
Execution strategies

Before Ansible 2, every task needed to be executed (and completed) on each machine before Ansible issued a new task to all machines. This meant that if you are performing tasks on a hundred machines and one of them is under-performing, all machines will go at the under-performing machine's speed.
With Ansible 2, the execution strategies have been made modular and therefore you can now choose which execution strategy you prefer for your playbooks. You can also write custom execution strategies, but this is beyond the scope of this book. At the moment (in Ansible 2.1) there are only three execution strategies: linear, serial, and free:

- Linear execution: This strategy behaves exactly as Ansible did prior to version 2. This is the default strategy.
- Serial execution: This strategy will take a subset of hosts (the default is five) and execute all tasks against those hosts before moving to the next subset and starting from the beginning. This kind of execution strategy could help you to work on a limited number of hosts so that you always have some hosts that are available to your users. If you are looking for this kind of deployment, you will need a load balancer in front of your hosts that needs to be informed about which nodes are in maintenance at every given moment.
- Free execution: This strategy will serve a new task to each host as soon as that host has completed the previous task. This will allow faster hosts to complete the playbook before slower nodes. If you choose this execution strategy you have to remember that some tasks could require a previous task to be completed on all nodes (for instance, clustering databases require all database nodes to have the database installed and running) and in this case they will probably fail.

**Ansible vault:**

Ansible vault is an exciting feature of Ansible that was introduced in Ansible version 1.5. This allows you to have encrypted passwords as part of your source code. A recommended practice is to NOT have passwords (as well as any other sensitive information such as private keys, SSL certificates, and so on.) in plain text as part of your repository because anyone who checks out your repository can view your passwords. Ansible vault can help you to secure your confidential information by encrypting and decrypting them on your behalf.

Ansible vault supports an interactive mode in which it will ask you for the password, or a non-interactive mode where you will have to specify the file containing the password and Ansible vault will read it directly.

For these examples, we will use the password ansible, so let's start creating a hidden file called .password with the string ansible in it. To do so, let's execute:

**Creating Encrypted Files:**

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.

**Encrypting Unencrypted Files**

If you have existing files that you wish to encrypt, use the ansible-vault encrypt command. This command can operate on multiple files at once:

```
$ ansible-vault encrypt foo.yml bar.yml baz.yml
```

**Decrypting Encrypted Files**

If you have existing files that you no longer want to keep encrypted, you can permanently decrypt them by running the ansible-vault decrypt command. This command will save them unencrypted to the disk, so be sure you do not want ansible-vault edit instead:

```
$ansible-vault decrypt foo.yml bar.yml baz.yml
```

**Viewing Encrypted Files**
If you want to view the contents of an encrypted file without editing it, you can use the ansible-vault view command:

```
$ ansible-vault view foo.yml bar.yml baz.yml
```

Example:

As below command for encrypt exiting file

```
$ ansible-vault encrypt webserverrole/vars/main.yml
New Vault password:
Confirm New Vault password:
```

```
webserverrole/vars/main.yml
$ANSIBLE_VAULT;1.1;AES256
39613935386338663630346330633134643466626332646132653562326231306366383564313237
30366365363138383732616536303861306161383265356 40a3563613563303931653132626538 30
34383432613433383637613766653238363731373165386137313737383139363537666432386635
3265363138363639630a653539373237386437656534626664663862373237323435393866313965
32353266656336303366306262343566666234353838393735303033326266333537353236346531
3764363834346431623765393262336337353532316665663163
```

As below command for `decrypt file`

```
$ ansible-vault decrypt webserverrole/vars/main.yml
Vault password:
```

```
webserverrole/vars/main.yml
---
# vars file for webserverrole
mysql=passwd
```

************************END*************************