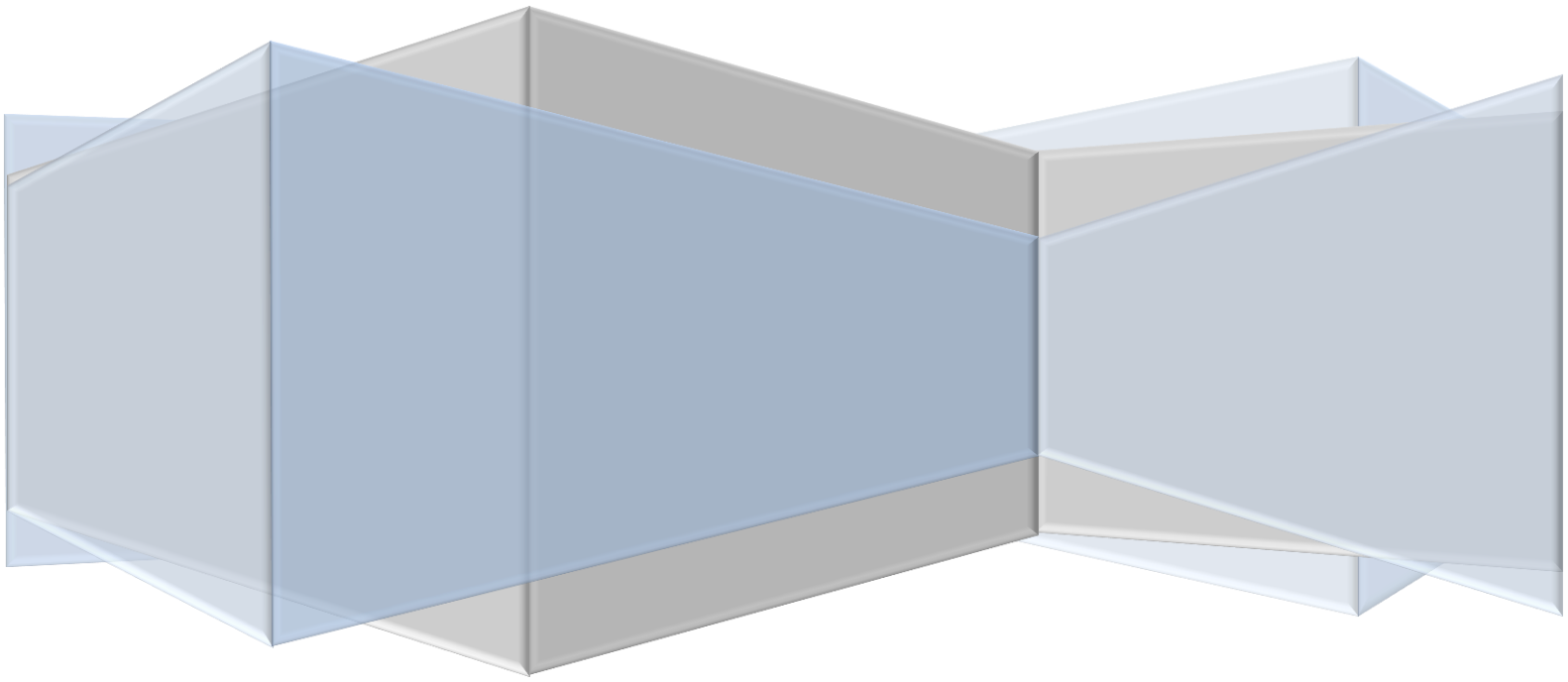
Automation Factory

Using Ansible

Ganesh Palnitkar



To start working with Ansible server and nodes, we have to make sure to comply pre-requisites, like establish ssh key sharing from Ansible server to nodes. To have Python running on the nodes.

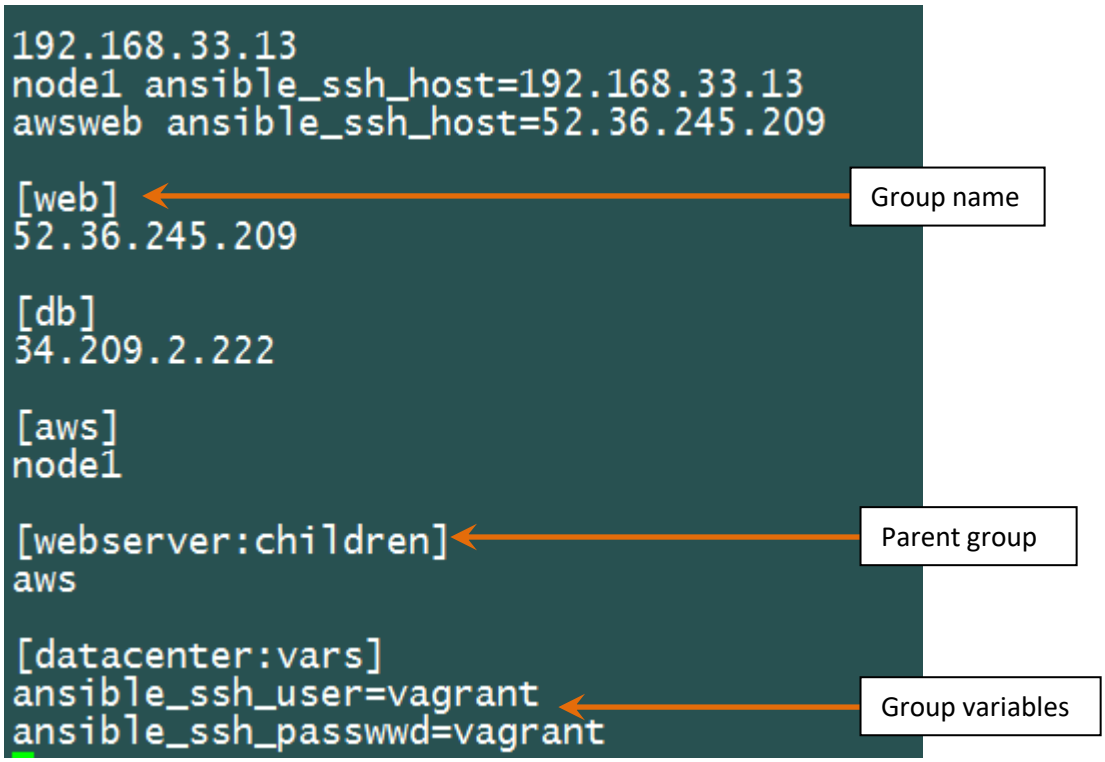
```
vagrant@ansibleserver:~/project$ ssh-keygen

vagrant@ansibleserver:~/project$ssh-copy-id –i ~/.ssh/id_rsa.pub vagrant@192.16.33.11
```

{ source of ssh key} **{Location of ssh key}** **{target user on remote node to copy public key**

Ansible Components:

Inventory file is the one in which we write all target (remote) servers that we want to manage using Ansible. This file can be updated manually or using a plugin, can be updated dynamically.

A sample inventory file looks like the one shown below.



Inventory file represented in YAML format

YAML format:

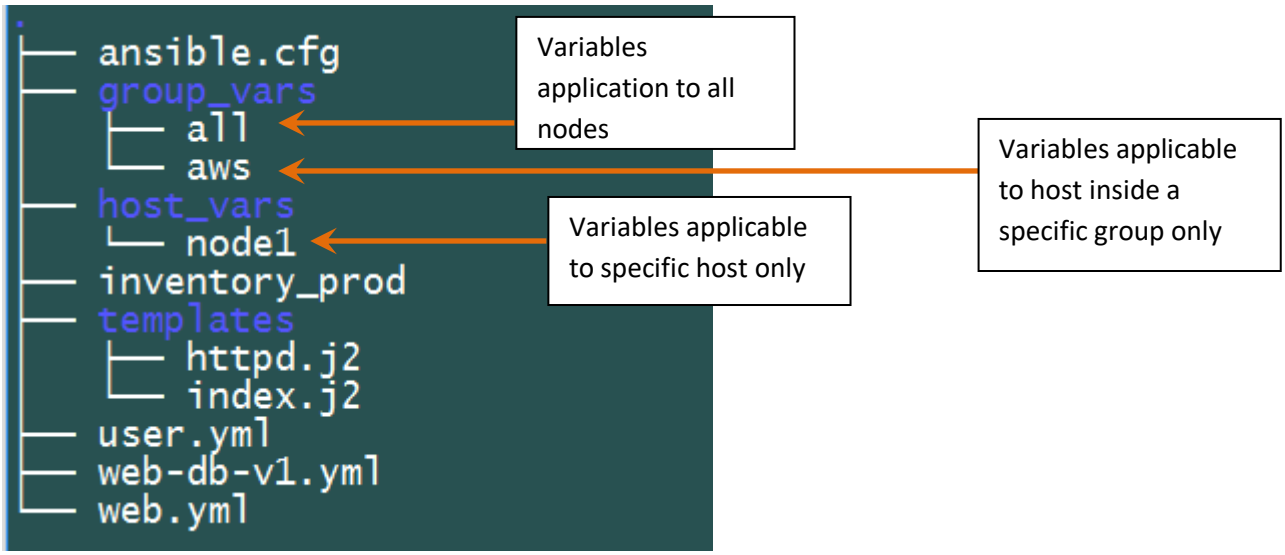
```yaml
all:
  hosts:
    192.168.33.13:
    node1.server:
    awsweb.server:
  children:
    web:
      hosts:
        52.36.245.209:
        192.168.33.13:
    db:
      hosts:
        34.209.2.222:
        node1.server:
        awsweb.server:
    maingrp:
      children:
        web:
        db:
    web:
      vars:
        ansible_ssh_user: vagrant
        ansible_ssh_passwd: vagrant
```

Inventory Management is important for effective and efficient use of Ansible to control your entire environment.

We can break it up in to the environments, like production and test etc. Thus, maintaining separate inventory files for specific environments.

Using **Variables** in **Ansible** inventory management will help in managing the inventory effectively.

This can be done by creating a folder structure as mentioned below.



Here variables mentioned in the 'all' file has the least precedence. Vars mentioned in the 'db' file which is the group_var directory, has the 2nd highest precedence and the variables mentioned in the web1 file inside the host_vars directory has the highest precedence.

Let's test this by using the user module to create a user using the 'username' variable mentioned inside these files.

Using user module to create user on remote node.

```
$ ansible webserver -i inventory_prod -m user -a "name={{username}}
password=12345" - sudo
```

In the 'all' var file update the below lines. Once updated run the user create module using command mentioned above.

```
- - -
# comment
username: ganesh
```

Update the 'aws' file with below lines. Once updated run the user create module using command mentioned above. Here name of the file must match with name of the group mentioned in the inventory file.

```
- - -
# comment
username: ganesh_aws
```

Update the 'node1' file with below lines. Once updated run the user create module using command mentioned above. Here the file name must match with the hostname for which we want to apply the variables.

```
- - -
# comment
username: ganesh_node1
```

Understanding the Ansible defaults:

To know about what all options can be set in the ansible configuration file, visit, www.docs.ansible.comand in 'getting started' look for configuration file details. Here we can see all options that can be set in the configuration file as defaults.

The options set in the ansible.cgf in the current directory have the least precedence. The option set in the environment variable has the highest precedence.

To test this, we can set the default key value, 'host_key_checking = False' in the ansible.cfgfile, and then try running the ping module on a target remote server. With this default key-value setting, we can override the requirement to check the host_key.

Few more default settings that we can modify and test are,

If we have python 3 installed on a specific remote server, we can update the inventory file and provide a behavioural pattern to be followed for that remote server. This patter is as mentioned below.

`192.168.33.12 ansible_python_interpreter=usr/bin/python2.7` … in this case the python 2.7 is installed in the 'usr/bin' directory. This can be set as per the specific system settings.

```
[defaults]

# some basic default values...

hostfile       = /etc/ansible/production/inventory_prod
library        = /usr/share/ansible
remote_tmp     = $HOME/.ansible/tmp
pattern        = *
forks          = 5
poll_interval  = 15
sudo_user      = root
#ask_sudo_pass = True
#ask_pass      = True
transport      = smart
remote_port    = 22
```

The 'ansible.cgf' file has ansible setting that one can modify to suit the environment.

This config file 'ansible.cgf' can be copied into the production folder and changes made in the file are then applied only for the playbooks located inside the folder.

Ansible Modules

- Core modules … modules supported by Ansible
- Extras … module updated created by community members and not supported by Ansible
- Deprecated … module that will be removed soon.

`$ ansible-doc -l`… to display all available core modules on ansible repo.
`$ ansible-doc<module name>…` man page for a module
`$ ansible-doc -s <name>…` help with some snippets on who to use a module inside a play book.
Core modules are categorized into multiple groups, like package deployment, network config, virtual machine, etc.
Common module that we can discuss are, 'copy' module, 'fetch', 'apt', 'yum', 'service' module etc.
Let's use the module to install webserver on a centos machine.
We will use the yum module and provide input parameters for the module to work. Using below command over command line,

```
$ ansible<hostname>-i hosts -m yum -a "name=httpd state=present" --become

$ ansible<hostname> -i hosts -m service -a "name=httpd enabled=yes state=started" --become
```

On the AWS DB node try running the yum module for package installation.

```
$ ansible<hostname> -i hosts -m apt -a "name=mariadb-server state=latest" --become

$ ansible<hostname> -i hosts -m service -a "name=mariadb state=started" --become
```

Understanding the setup module
'Setup' module is a module to gather facts from a system.

```
$ ansible web -i hosts -m setup -a "filter=ansible_os_family"
```

Module usage with example:
File Module:

```
$ ansible web -i inventory -m file -a "name=<path/filename> state touch"

$ ansible web -i inventory -m file -a "path=<path/filename> state touch"
```

Copy Module:

```
$ ansible web -i inventory -m copy -a "content<html><h1>Hello world</h1></html> dest=/var/www/html/index.html"
```

Template Module

```
$ ansible web -i inventory -m template -a "src=<path/file.j2>dest=<filepath>"
```

User module

```
$ ansible web - i inventory –m user –a “name=ganeshhp comment=Ganesh Palnitkar
gid=4234 uid=4010”
```

Package Modules

```
$ ansible web – i inventory –m yum –a “name=ntp state=latest”

$ ansible web – i inventory –m apt –a “name=apache2 state=latest”
```

Service module

```
$ ansible web –I inventory –m service –a “name=apache2 state=started enabled=yes”
```

`$ ansible<hostname> -m shell –a <shell command> --become` ….. Here we can pass a shell command to be executed on a remote machine.

Host /group Target pattern

group1:group2 … grp1 OR grp2
!group1 …. Not grp1
web*.autofact.com ….. wildcard
group1:&group2 … host machines that are only common with both groups only be applied with the change.

Collecting Facts on remote system

`$ ansible<hostname> -i inventory –m setup –a “filter=ansible_eth*”` …. This will run the ohai profiler on the remote server and gather facts and return those to the ansible server.

The command and shell modules are the only modules that just take a list of arguments and don’t use the key=value form. This makes them work as simply as you would expect:

```
tasks:
    -   name: enable selinux
        command: /sbin/setenforce 1
```

The command and shell module care about return codes, so if you have a command whose successful exit code is not zero, you may wish to do this:

```
tasks:
    -   name: run this command and ignore the result
        shell: /usr/bin/somecommand || /bin/true
```

Or this:

```
tasks:
    -   name: run this command and ignore the result
        shell: /usr/bin/somecommand
        ignore_errors: True
```

If the action line is getting too long for comfort you can break it on a space and indent any continuation lines:

```
tasks:
    -   name: Copy ansible inventory file to client
        copy:
          src: /etc/ansible/hosts
          dest: /etc/ansible/hosts
          owner: root
          group: root
          mode: 0644
```

Variables can be used in action lines. Suppose you defined a variable called `vhost` in the `vars` section, you could do this:

```
tasks:
    -   name: create a virtual host file for{{vhost}}
        template:
          src: somefile.j2
          dest: /etc/httpd/conf.d/{{vhost}}
```

Play and Playbooks

- Plays help to map the hosts to tasks.
- A play can have multiple tasks
- A playbook can have multiple plays.

A sample playbook, each coloured rectangle represents a play. Each play is mapped to the host or a group, parent group etc.

```
---
- hosts: webservers
  remote_user: root
  tasks:
  - name: Install Apache
    yum: name=httpd state=present
  - name: Start Apache
    service: name=httpd state=started

- hosts: dbservers
  remote_user: root
  tasks:
  - name: Install MySQL
    yum: name=mysql-server state=present
  - name: Start MySQL
    service: name=mysqld state=started
```

Play

Play

Using white space / indentation is very specific and has to be followed while writing the play / playbook.

```
---
- hosts: webservers
  remote_user: root
  tasks:
  - name: Install Apache
    yum: name=httpd state=present
  - name: Start Apache
    service: name=httpd state=started
```

Tasks are executed in the order – top down. Thus we have to be careful while specifying the tasks in the play.

Tasks use modules

You can also control the order in which hosts are run. The default is to follow the order supplied by the inventory:

```
- hosts: all
  order: sorted
  gather_facts: False
```

```
    var: inventory_hostname
    tasks:
    - debug:
```

Possible values for order are:

inventory:

The default. The order is ‘as provided’ by the inventory

reverse_inventory:

As the name implies, this reverses the order ‘as provided’ by the inventory

sorted:

Hosts are alphabetically sorted by name

reverse_sorted:

Hosts are sorted by name in reverse alphabetical order

shuffle:

Hosts are randomly ordered each run

To execute a playbook use the command,

```
$ ansible-playbook<playbook.yml>
```

Optional parameters to pass while executing playbook,

```
$ ansible-playbookplaybook.yml --step
$ ansible-playbookplaybook.yml --limit playbook.retry
```

A sample playbook, note the indentation and syntax used for writing tasks and modules.

```
-   hosts: webserver
    remote_user: root
    become: yes
    tasks:
    - name: ensure apache is at the latest version
      yum: name=httpd state=present
    - name: start the apache service
      service: name=httpd state=started enabled=yes

-   hosts: dbserver
    remote_user: root
    become: yes
    tasks:
    - name: ensure MySQL is installed
      apt: name=mariadb-server state=present
    - name: ensure that MySQL service is started
      service: name=mariadb state=started
```

We can verify the Playbook syntax using the Lint utility.

```
$ ansible-lint verify-apache.yml
```
…here verify-apache.yml is a playbook.

One can also call a playbook inside a play., like shown below.,

```
---
- hosts: webserver
  tasks:
  - include: common.yml
  - include: web.yml
```

Notify Function can be used in the Playbook that calls the Handler. Handlers are list of Tasks listed under the tag of handlers that are invoked only on a certain condition in the execution of tasks.

```
handlers:
    -   name:restart memcached
        service:
           name:memcached
           state:restarted
    -   name:restart apache
        service:
           name:apache
           state:restarted
```

For failed task in the playbook execution one can use below command so as Ansible will confirm on running a particular task before executing it.

```
$ ansible-playbookplaybook.yml --step.
```

The step asks you on before executing each task and you could choose (N)o/(y)es/(c)ontinue.

Roles:

In order to create roles, create a Directory named as Role inside the Ansible directory.

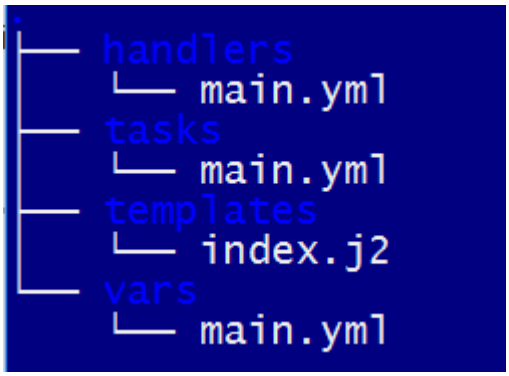
The role folder structure can also be created using the ansible-galaxy utility.

About ansible-galaxy

```
$ ansible-galaxy init <role name>
```

This command will create a default folder structure with required files placed in it.

As shown below create directories and files inside the roles directory. Here ‘webserver’ is the role.



The tasks will be located in the tasks directory and written in main.yml file.

Roles hold the tasks inside the tasks folder in the main.yml file. So the main.yml file inside the tasks folder would appear as shown below.,

```
---
- name: ensure Apache is installed
  apt: name=apache2 state=present

- name: starte the apache service
  service: name=apache2 state=started

- name: Copy site file
  template: src=index.j2 dest={{ doc_root }}/index.html
  notify:
  - Restart Apache
```

The file will only contain tasks and no hosts statement or vars definitions, etc.

The ‘vars’ folder thus has the main.yml file to store all variables for the role.

```
http_port: 80
doc_dir: /var/www/html/ansible/
doc_root: /var/www/html/
username: ganeshhp
```

Similar to this the template folder would contain the template file, like index.j2, etc. and main.yml file inside ‘handler’ folder will have action statement as notified in ‘notify’ command.

In order to call the role inside a playbook we can use below syntax. The playbook in such case will be located outside the ‘roles’ folder. And will typically have syntax similar to one mentioned below.

```
- hosts: web
  sudo: yes
  roles:
  - webserver
```

Here in this file we are calling the role with the 'roles' statement and then mentioned the role name.

Ansible Galaxy for remote Role repository

https://galaxy.ansible.com

Ansible provides a repository of ready-to-use roles for almost all requirements that you can think of.

Just pull the role to your Ansible controller and start using.

On the Galaxy web link we can explore all different roles categorized as, 'Most starred', 'Most watched', 'Most Downloaded', etc.

To install a particular role on to the Ansible control server, we can use the command,

Some ansible-galaxy commands

```
$ ansible-galaxy { init, remove, delete, list, search, import, setup, login, info, install}

$ ansible-galaxy install <role name>
```

How to secure file or a string.

Ansible Vault

```
$ ansible-vault [create|decrypt|edit|encrypt|encrypt_string|rekey|view] [options] [vaultfile.yml]
```

Common Options

`--ask-vault-pass`

ask for vault password

`--new-vault-id <NEW_VAULT_ID>`

the new vault identity to use for rekey

`--new-vault-password-file`

new vault password file for rekey

`--vault-id`

the vault identity to use

`--vault-password-file`

vault password file

`--version`

show program's version number and exit

`-h, --help`

show this help message and exit

`-v, --verbose`
verbose mode (-vvv for more, -vvvv to enable connection debugging)

To create a new encrypted key file
`$ ansible-vault create abc.yml`(a file abc.yml is create which will have encrypted contents stored)

To encrypt existing file
```
$ ansible-vault encrypt existing.yml
```

To update or rekey files
```
$ ansible-vault rekey abc.yml
```

Editing an encrypted file

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

Viewing an encrypted file
```
$ ansible-vault view abc.yml
```

Decrypting files
```
$ ansible-vault decrypt abc.yml
```

To use an encrypted file while executing a playbook.
```
$ ansible-playbook -i hosts abc.yml --ask-vault-pass
```

This will prompt the user for supplying vault password.

Variable_Prompt

When running a playbook, you may wish to prompt the user for certain input, and can do so with the ‘vars_prompt’ section.
A common use for this might be for asking for sensitive data that you do not want to record.

Here’s an example…
in this example, the variable keyname is the value for name prompt.

```
---
- hosts: all
  vars_prompt:

    - name: username
      prompt: "What is your username?"
      private: no

    - name: password
```

```
    prompt: "What is your password?"

  tasks:

    - debug:
        msg: 'Logging in as {{ username }}'

    - user:
        name: '{{ username }}'
        password: '{{ password }}'
        state: 'present'
```

Ignoring Failed Commands

Generally playbooks will stop executing any more steps on a host that has a task fail. Sometimes, though, you want to continue on. To do so, write a task that looks like this:

```
    - name: This will not be counted as a failure
      command: '/bin/yum install none'
      ignore_errors: yes
```

We can also define the criteria to set for identifying failure in logs search.
You may check for failure by searching for a word or phrase in the output of a command:

```
- name: Fail task when the command error output prints FAILED
  command: /usr/bin/example-command -x -y -z
  register: command_result
  failed_when: "'FAILED' in command_result.stderr"
```

or based on the return code:

```
- name: Fail task when both files are identical
  raw: diff foo/file1 bar/file2
  register: diff_cmd
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

Conditional Statements:

When statement:

```
tasks:
    -   name: "take an action on certain conditional fulfilment"
        when: ansible_facts['os_family'] == "RedHat"
        command: /sbin/shutdown –t now
```

or we can have conditional check in a group as statement below.

```
tasks:
  - name: "shut down CentOS 6 and Debian 7 systems"
    command: /sbin/shutdown -t now
    when: (ansible_facts['distribution'] == "CentOS" and ansible_facts['distribution_major_version'] == "6") or
          (ansible_facts['distribution'] == "Debian" and ansible_facts['distribution_major_version'] == "7")
```

Using conditional when with loop:

```
tasks:
    - command: echo {{ item }}
      loop: [ 0, 2, 4, 6, 8, 10 ]
      when: item > 5
```

Applying ‘when’ conditional statement to roles and include.

```
- import_tasks: other_tasks.yml # note "import"
  when: x is not defined
```

```
hosts: webservers
  roles:
     - role: debian_stock_config
       when: ansible_facts['os_family'] == 'Debian'
```

Register variables: Register is default variable that can be initialized dynamically during command, module execution.

```
- name: test play
  hosts: all

  tasks:

      - shell: cat /etc/motd
        register: motd_contents

      - shell: echo "motd contains the word hi"
        when: motd_contents.stdout.find('hi') != -1
```

or, we can validate ‘register’ variable contents for emptiness.

```
- name: check registered variable for emptiness
  hosts: all

  tasks:

      - name: list contents of directory
        command: ls mydir
        register: contents

      - name: check contents for emptiness
        debug:
          msg: "Directory is empty"
        when: contents.stdout == ""
```

Tags:

If we have large number of tasks and you want to selectively run a task or number of tasks or a play then we can make use tags to selectively execute a task or a play.

```
tasks:
- yum:
    name:
    - httpd
    - memcached
    state: present
  tags:
  - packages

- template:
```

```
    src: templates/src.j2
    dest: /etc/foo.conf
  tags:
  - configuration
```

Once we have tagged the task or play , we can call the task or play by using its tag identity.

```
$ ansible-playbook playbook.yml --tags "configuration,packages"
```
or,

```
$ ansible-playbook example.yml --skip-tags "packages"
```

Few handy commands on Ansible...

```
$ ansible-doc -t <type_of_module_from_list> --list
```

Types of docs we can list...

```
{become,cache,callback,cliconf,connection,httpapi,inventory,lookup,netconf
,shell,module,strategy,vars}]
```

Below commands can help to list all default config values that are defined in the Ansible.cfg file.

```
$ ansible-config -version {list dump view}
```

Using inventory command to list all devices from inventory file.

```
$ ansible-inventory --list
```

Using Ansible for Windows management.

Pre-Requisites before we get started for managing Windows hosts using Ansible.

- Supported Microsoft Windows operating system versions:
  - Windows Server (2008, 2008 R2, 2012, 2012 R2, 2016, or 2019)
  - Windows 7, 8.1, or 10
- To manage a Windows-based server, Ansible must connect and run code
  - WinRM must be enabled
  - Ansible must be able to authenticate to the managed host
  - The managed host must have PowerShell 3.0 or newer and .NET Framework 4.0 or newer (Windows Server 2012 and later and Windows 8.1 and later has the right software pre-installed)

On the Ansible Control server install the python package, pywinrm using below command.

```
$ sudo yum install python-pip
```

```
$ sudo pip install pywinrm
```

Update the inventory file on the Ansible control server by adding the windows machine entry.

```
[windows]
172.31.42.206

[windows:vars]
ansible_connection=winrm
ansible_user=<username>
ansible_password=<password>
ansible_winrm_server_cert_validation=ignore
ansible_winrm_transport=basic
ansible_winrm_port=5985
```

The above variable declaration can also be done inside 'group_vars' folder by creating variable file named after the group name… e.g. group_vars → windows

Now, on the windows machine that we want to manage using Ansible, make few changes as show below.

```
ps> winrm quickconfig-q

ps> winrm set winrm/config/service '@{AllowUnencrypted="true"}'

ps> winrm set winrm/config/service/path '@{Basic="true"}'
```