DevOps

Caltech | Center for Technology & Management Education

Post Graduate Program in DevOps

simplilearn

Ansible Implementation

Learning Objectives

By the end of this lesson, you will be able to:

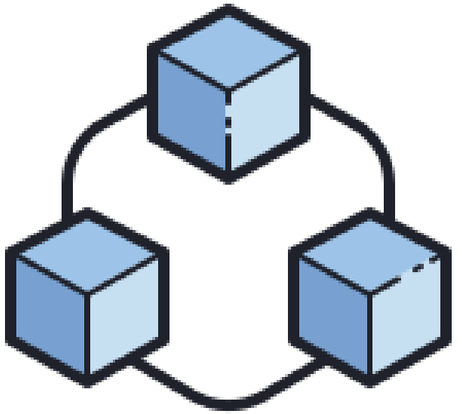
- Explain Ansible modules
- Illustrate and create Ansible roles
- Define the working of Ansible galaxy
- Work with Ansible tower and conditionals

Ansible Modules

Ansible Module

Ansible module is the smallest piece of code or command that runs on the node or client machine.

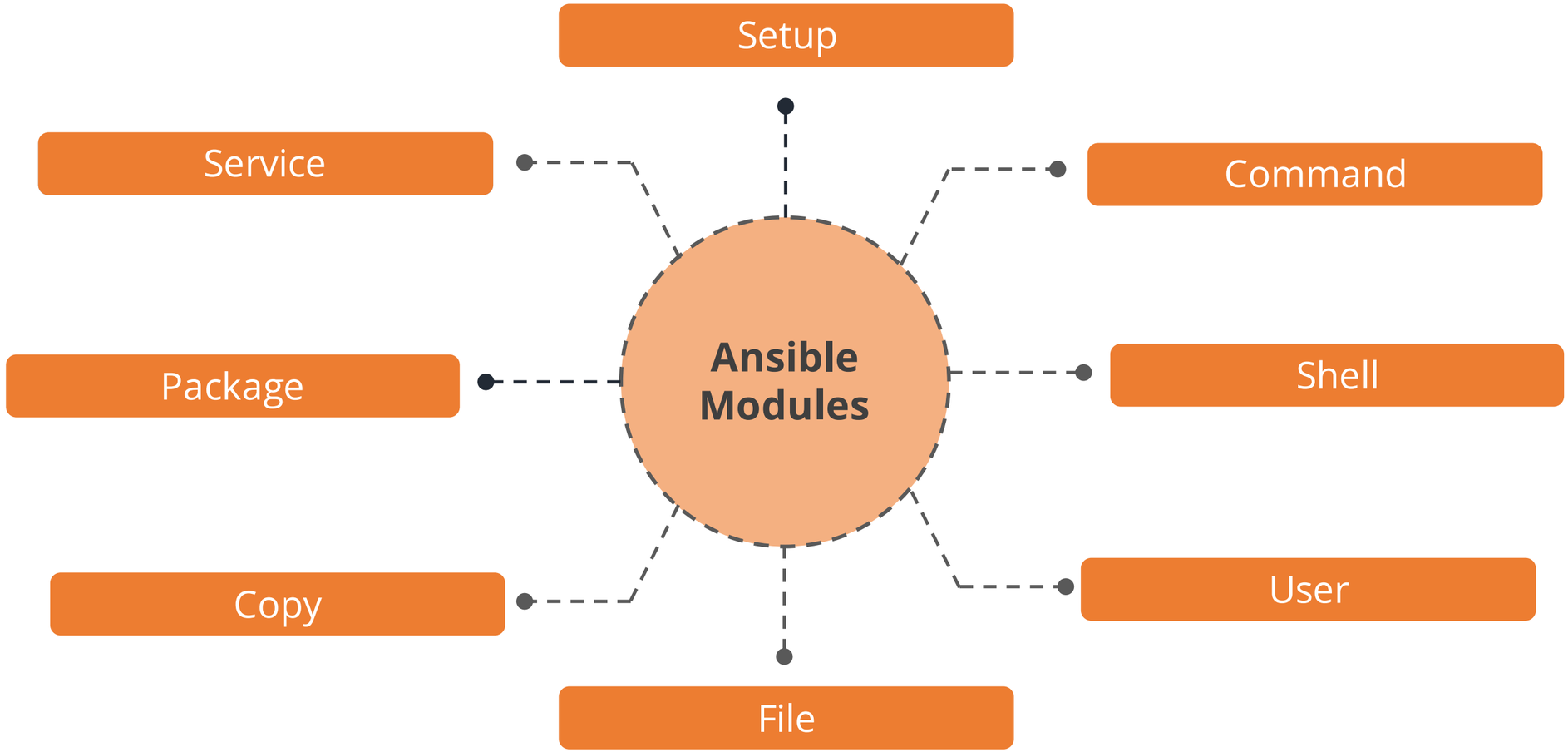
- Modules are the building blocks of Ansible.
- Modules can be used as a part of a playbook or on their own as a commands.
- Syntax to implement a module is shown below:



```
ansible <host OR group> -m <module_name> -a <"arguments"> -u <username> --become
```

Types of Modules

Ansible provides a huge library of modules for a user. Some of the most frequently used modules are given below:



Types of Modules

Setup Module:

- Setup module gathers the hardware, network, and OS information of the target machines.
- Execute the below command to fetch system information of node machine:

```
$ ansible webservers –m setup
```

Command Module:

- Command module executes a specific command on the target machine.
- Some frequently used commands are given below:

```
$ ansible webservers –m command - an ‘uptime’
$ ansible webservers –m command –a ‘hostname’
```

Types of Modules

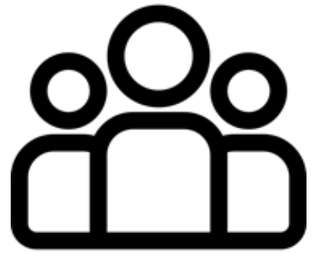
Shell Module:

- Shell module commands are run in **/bin/sh** shell.
- The difference between the shell and command module is that shell module allows a user to run operators as an additional feature.
- Example of a shell module command is given below:

```
$ ansible webservers -m shell -a 'ls -l > temp.txt'
```

User Module:

- This module is used to create or delete users.
- To add user, execute the command given below:



```
$ ansible webservers -m user -a 'name=user1 password=user1' --become
```

- To delete user, execute the command given below:

```
$ ansible webservers -m user -a 'name=user1 state=absent' --become
```

Types of Modules

File Module:

- This module is used to create files, directories, change file permissions, and ownership.
- To create a file, execute the below command:

```
$ ansible webservers -m file -a 'dest=<destination location> state=touch mode=600 owner=ansible group=ansible'
```

- To create a directory using the file module, you need to set two parameters and run the command as shown below:

1. Path(alias: name, dest)
1. State

```
$ ansible webservers -m file -a "dest=/home/ansible/vndir state=directory mode=755"
```

Types of Modules

- To delete a file, execute the command given below:

```
$ ansible webservers -m file -a "dest= <destination of file> state=absent"
```

- To delete a directory, execute the command given below:

```
$ ansible webservers -m file -a "dest=/home/ansible/vndir state=absent"
```

Copy Module:

- This module is used to copy files to the target machines.
- To copy a file, execute the command given below:

```
$ ansible webservers -m copy -a "src=sample.txt dest=/home/ansible/sample.txt"
```

Types of Modules

Managing Software Packages:

- Package modules are used to update existing packages, or add or remove softwares.
- To install software through **yum** or **apt,** use the below commands:

Installing GIT

```
$ ansible webservers –m yum -a “name=git state=present” --become
```

Checking version

```
$ ansible webservers -m yum -a “name=git state=latest”
```

Installing Apache Web Server

```
$ ansible webservers -m yum -a "name=httpd state=present" –become
```

Checking Maven

```
$ ansible webservers -m yum -a "name=maven state=absent" –become
```

Types of Modules

Managing Services Module:

- This module is used to manage system services with Ansible.

Starting a service

```
$ ansible webservers -m service -a "name=httpd state=started" --become
```

Stopping a service

```
$ ansible webservers -m service -a "name=httpd state=stopped" --become
```

Restarting a service

```
$ ansible webservers -m service -a "name=httpd state=restarted --become
```

Working with Ansible Modules

The different ways to include modules in a playbook are given below:

1) Within playbook:

```
- name: reboot the servers
  command: /sbin/reboot -t now
```

1) As arguments using YAML syntax:

```
- name: restart webserver
  service:
    name: httpd
    state: restarted
```

- It is also called complex args.

- All modules return **JSON** format data.
- Modules should be idempotent.
- Modules can trigger **change events** using **handlers** to run any extra tasks.

Assisted Practice

Ansible Modules

Problem Statement:

Create an Ansible program to implement modules.

Assisted Practice: Guidelines

Steps to perform:

1. Install Ansible
2. Setup webserver
3. Create a playbook containing modules
4. Execute the playbook

Ansible Facts

Ansible Facts

Ansible facts are system properties collected by Ansible while executing tasks on a machine.

- The facts contain details about the storage and network configuration of target machine.
- These details are used during Ansible playbook execution to perform runtime decisions.
- Facts work majorly with conditionals and playbooks for accelerating automation process.

```
[user@instructor ~]$ ansible localhost -m setup
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "10.0.2.15",
            "192.168.122.1"
        ],
        "ansible_all_ipv6_addresses": [
            "fe80::f4ef:a64b:e19c:cfc1"
        ],
        "ansible_apparmor": {
            "status": "disabled"
        },
        "ansible_architecture": "x86_64",
        "ansible_bios_date": "12/01/2006",
        "ansible_bios_version": "VirtualBox",
        "ansible_cmdline": {
            "BOOT_IMAGE": "/vmlinuz-3.10.0-862.14.4.el7.x86_64",
            "LANG": "en_US.UTF-8",
            "crashkernel": "auto",
            "quiet": true,
            "rd.lvm.lv": "centos/swap",
            "rhgb": true,
            "ro": true,
            "root": "/dev/mapper/centos-root"
```

Assisted Practice

Ansible Facts

Problem Statement:

Create an Ansible program to implement facts.

Assisted Practice: Guidelines

Steps to perform:

1. Setup webserver
2. Create a playbook containing facts
3. Execute the playbook

Ansible Loops and Conditionals

Ansible Loops

Ansible loop is a block of code used to repeat tasks.

- Ansible loops include changing ownership on several files and/or directories with the file module, creating multiple users, and repeating a step.
- Ansible offers two keywords for creating loops:

1. loop
2. with_<lookup>.

Working with Loops

- The **with_<lookup>** is dependent on lookup plugins.
- The loop keyword is similar to **with_list**.
- The loop keyword will not accept a string as input.
- Use **with_items** to performs implicit single-level flattening.
- It is recommended to use flatten(1) with loop for accurate output.
- For example, to get the output as:

```
with_items:
  - 1
  - [2,3]
  - 4
```

you would need:

```
loop: "{{ [1, [2,3] ,4] | flatten(1) }}"
```

Working with Loops

Standard loops:

Iterating over a simple list

- You can define the list directly in the task:

```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
     - testuser1
     - testuser2
```

- Define list in a variable file and refer to the name of the list in the task.
- You can also define list in **vars** section of your play.
- Use the below syntax to refer a list in a variable file or **vars** section:

```
loop: "{{ somelist }}"
```

Working with Loops

Iterating over a list of hashes:

- With list of hashes, refer the sub keys in a loop.

For example:

```
- name: add several users
  user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

Working with Loops

Iterating over a dictionary:

- To loop over a dict, use the **dict2items** dict filter as shown below:

```
- name: create a tag dictionary of non-empty tags
  set_fact:
    tags_dict: "{{ (tags_dict|default({}))|combine({item.key: item.value}) }}"
  loop: "{{ tags|dict2items }}"
  vars:
    tags:
      Environment: dev
      Application: payment
      Another: "{{ doesnotexist|default() }}"
  when: item.value != ""
```

Working with Loops

Complex loops:

Iterating over nested lists:

- Use Jinja2 expressions to iterate over complex lists.
- Below is an example of nested loop:

```
- name: give users access to multiple databases
  mysql_user:
    name: "{{ item[0] }}"
    priv: "{{ item[1] }}.*:ALL"
    append_privs: yes
    password: "foo"
  loop: "{{ ['alice', 'bob'] |product(['clientdb', 'employeedb', 'providerdb'])|list }}"
```

Working with Loops

Below are some variables that are used along with loops to execute different operations:

1. **register:** To register a variable using loops
1. **until:** To repeat a task until a condition is met
1. **ansible_play_batch:** To use loop with inventory
1. **label** and **loop_control:** To limit the output of the loop
1. **pause:** To stop the execution for a short time period between different tasks

Working with Loops

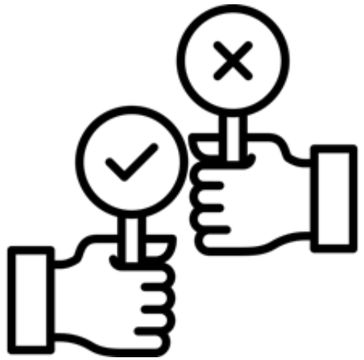
Below is the list of variables used using **extended** option with loop control:

Variable	Description
ansible_loop.allitems	The list of all items in the loop
ansible_loop.index	The current iteration of the loop (1 indexed)
ansible_loop.index0	The current iteration of the loop (0 indexed)
ansible_loop.revindex	The number of iterations from the end of the loop (1 indexed)
ansible_loop.revindex0	The number of iterations from the end of the loop (0 indexed)
ansible_loop.first	True if it is the first iteration of the loop
ansible_loop.last	True if it is the last iteration of the loop
ansible_loop.length	The number of items in the loop
ansible_loop.previtem	The item from the previous iteration of the loop; Undefined during the first iteration
ansible_loop.nextitem	The item from the following iteration of the loop; Undefined during the last iteration

Ansible Conditionals

Ansible conditionals are the control statements used in a playbook in order to generate accurate report or output.

- The result of a play may depend on the variable, fact, or previous task result.
- Conditional statements are used in playbooks where there are multiple variables representing different entities such as software packages and servers.
- Conditional statements help in controlling the flow of execution.
- With the help of conditional statements, one can determine the tasks which can be skipped or to be run on particular nodes.



Ansible Conditionals

There are multiple ways to use conditional statements in a playbook. However, below are the four major ways of implementing Ansible conditionals:

When statement

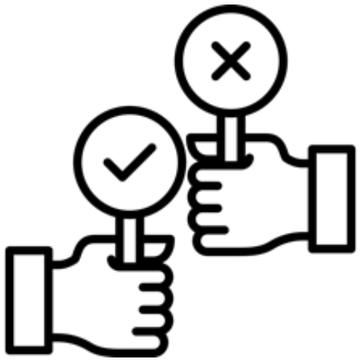
Custom facts

Conditional imports

Loops

Ansible Conditionals

When Statement:



- When statement is mainly used to skip a process or task.
- **when** clause contains a raw **Jinja2** expression without double curly braces.
- Below is an example of **when** clause:

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

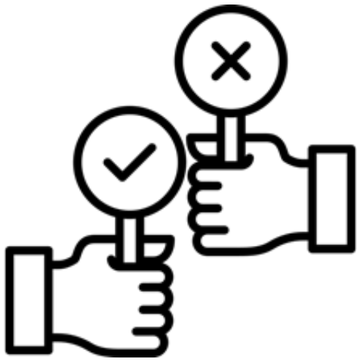
- You can also use parentheses to group conditions as shown below:

```
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")
```

Ansible Conditionals

Filters with when clause:

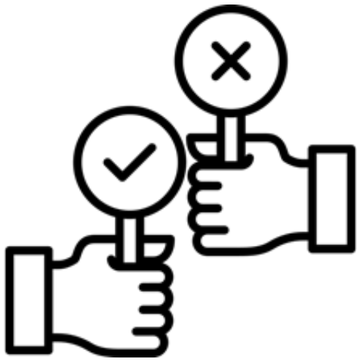
- Filters can be coupled with **when** clause to direct the execution of the play.
- An example to ignore the error of one statement and execute following statements based on success or failure is shown below:



```
tasks:
  - command: /bin/false
    register: result
    ignore_errors: True

  - command: /bin/something
    when: result is failed

  # In older versions of ansible use ``success``, now both are valid but succeeded uses the correct tense.
  - command: /bin/something_else
    when: result is succeeded

  - command: /bin/still/something_else
    when: result is skipped
```

Ansible Conditionals

With loops:



- Process the conditional statement independently for each item in the loop
- An example of **when** clause with loop is shown below:

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

- To skip the whole task depending on the loop variable, use the **|default** filter to provide an empty iterator as shown below:

```
- command: echo {{ item }}
  loop: "{{ mylist|default([]) }}"
  when: item > 5
```

- Below is an example of using a dict in a loop:

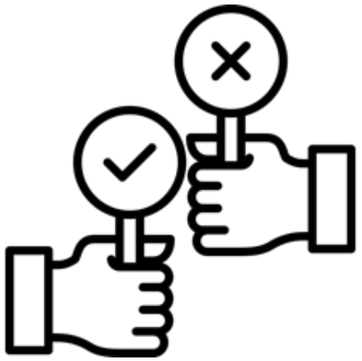
```
- command: echo {{ item.key }}
  loop: "{{ query('dict', mydict|default({})) }}"
  when: item.value > 5
```

Ansible Conditionals

Loading in custom facts:

- Below is an example of **when** clause with custom fact:

```
tasks:
    - name: gather site specific fact data
      action: site_facts
    - command: /usr/bin/thingy
      when: my_custom_fact_just_retrieved_from_the_remote_system == '1234'
```



When clause with roles:

- It is recommended to group multiple tasks if they share the same conditional statement.

- In such cases, all the tasks get evaluated, but the conditional is applied to every task as shown below:

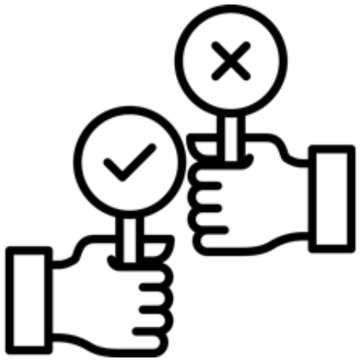
```
- import_tasks: tasks/sometasks.yml
  when: "'reticulating splines' in output"
```

Ansible Conditionals

Conditional Imports:

- Running one playbook on multiple platforms is one of the situations where conditional imports are used.

- As an example, the name of the Apache package may be different between CentOS and Debian as shown below:



```
---
- hosts: all
  remote_user: root
  vars_files:
    - "vars/common.yml"
    - [ "vars/{{ ansible_facts['os_family'] }}.yml", "vars/os_defaults.yml" ]
  tasks:
  - name: make sure apache is started
    service: name={{ apache }} state=started
```

Assisted Practice
Ansible Loops

Problem Statement:

Create an Ansible program to implement loops.

Assisted Practice: Guidelines

Steps to perform:

1. Set up server
2. Create a playbook containing loops
3. Execute the playbook

Assisted Practice
Ansible Conditionals

Problem Statement:

Create an Ansible program to implement conditional statements.

Assisted Practice: Guidelines

Steps to perform:

1. Set up server
2. Create a playbook containing conditional statements
3. Execute the playbook

Ansible Roles

Ansible Roles

Ansible Role is a framework that applies certain collections of variables, tasks, files, templates, and modules based on a directory structure.

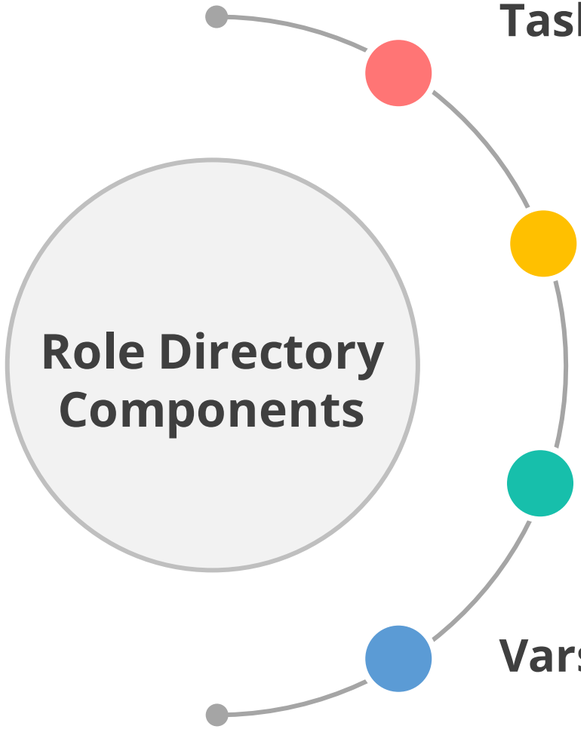
- Grouping the content by roles, tasks can be easily assigned to specified hosts.
- Role is the basic process for breaking a playbook into multiple files.
- It is limited to a particular functionality or desired output.
- It contains all the necessary steps to provide that result.
- Roles are not playbooks and cannot be executed directly.

Role Directory Structure

```
site.yml
webservers.yml
fooservers.yml
roles/
    common/
        tasks/
        handlers/
        files/
        templates/
        vars/
        defaults/
        meta/
    webservers/
        tasks/
        defaults/
        meta/
```

Role Directory Components

The different directory components for a role are given below:



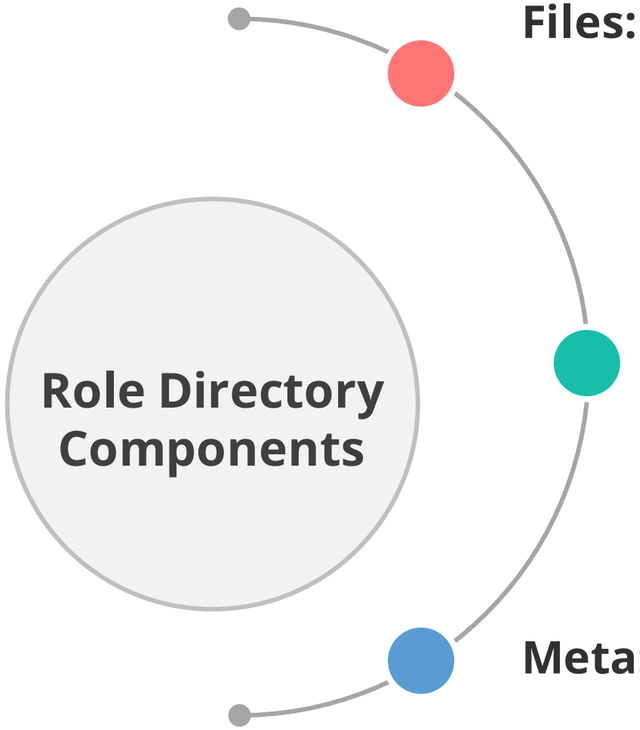
Tasks: Contains the list of processes to be executed

Handlers: Contains notify handlers which will be used within or outside a role

Default: Contains the variables for the current role

Vars: Contains external variables for the current role

Role Directory Components



Files: Contains the list of files to be deployed by the role

Templates: Contains the templates to be executed by the role

Meta: Contains general information about the role

Role Execution Flow

The order of execution for playbook containing roles is given below:

1. Pre_tasks defined in the play
1. Handlers triggered so far will be run
1. Each role listed in roles will execute in turn
1. Tasks defined in the play
1. Handlers triggered so far will be run
1. Post_tasks defined in the play
1. Handlers triggered so far will be run

Creating a Role Directory

The command to create a role directory using Ansible galaxy is given below:

```
$ ansible-galaxy init --force --offline ansirole
- ansirole was created successfully

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

8 directories, 8 files
```

Inline Roles

- Roles can be used inline with any other tasks using **import_role** or **include_role** as shown below:

```
---
- hosts: webservers
  tasks:
    - debug:
        msg: "before we run our role"
    - import_role:
        name: example
    - include_role:
        name: example
    - debug:
        msg: "after we ran our role"
```

Role Duplication

```
---
- hosts: webservers
  roles:
    - foo
    - foo
```

- In the above example, the role foo will only be run once as the parameters defined are not different for every definition.

- To make roles run more than once, there are two options:

1. Pass different parameters in each role definition
2. Add **allow_duplicates: true** to the **meta/main.yml** file for the role

```
---
- hosts: webservers
  roles:
    - role: test1
      vars:
        message: "first"
    - { role: test1, vars: { message: "second" } }
```

- In this example, because each role definition has different parameters, **test1** will run twice.

Role Default Variables

- Role default variables allow you to set default variables for included or dependent roles.
- To create a default variable, add a **defaults/main.yml** file in your role directory.
- They have the lowest priority of any variables available.
- They can be overridden by other variables like inventory variables.

Role Dependencies

- Role dependencies automatically pull in other roles when using a role.
- They are stored in the **meta/main.yml** file.
- They contain a list of roles and parameters to insert before the specified role as shown below:

```
---
dependencies:
  - role: common
    vars:
      some_parameter: 3
  - role: apache
    vars:
      apache_port: 80
  - role: postgres
    vars:
      dbname: blarg
      other_parameter: 12
```

Role Search Path

Ansible searches roles in two different ways as shown below:

1. In **roles/ directory**
2. In **/etc/ansible/roles**

Ansible Galaxy

What Is Ansible Galaxy?

Ansible galaxy is a website where users can share roles to a command-line tool for installing, creating, and managing roles.

- Ansible galaxy consists of pre-made roles that can be downloaded in the local machine.
- It reduces the effort to create roles from scratch and is useful in sharing custom roles with the Ansible community.

Ansible Galaxy Commands

Some of the Ansible Galaxy commands are given below:

- To display the list of installed roles, with version numbers

```
ansible-galaxy list
```

- To remove an installed role

```
ansible-galaxy remove [role]
```

- To create a role template suitable for submission to Ansible Galaxy

```
ansible-galaxy init [role]
```

Creating Collections

ansible-galaxy-collection command implements the following commands:

1. **init:** Creates a structure of collection based on the default template

1. **build:** Creates a collection artifacts to be uploaded to the galaxy or a repository

1. **publish:** Publishes a built connection to the galaxy

1. **install:** Installs one or more connections

Ansible Tower

Introduction to Ansible Tower

Ansible Tower is a web-based solution for managing applications.

- Provides a dashboard with the status of all the hosts
- Shares the **SSH** credentials without exposing them, logging jobs, manage inventories, and integrating cloud providers
- Allows user to perform real-time node monitoring
- Is a dashboard with NOC-style display for Ansible environment

Prerequisites of Ansible Tower

1. The following operating systems support Ansible Tower:

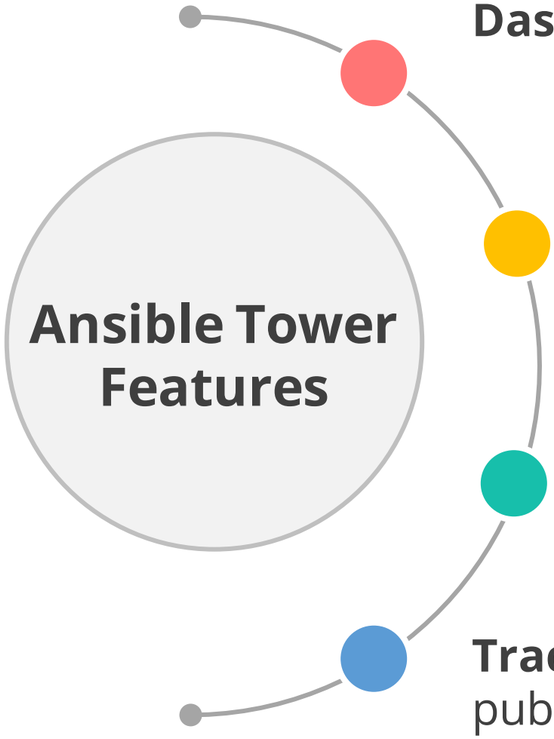
- RedHat Enterprise Linux 6 64-bit
- RedHat Enterprise Linux 7 64-bit
- CentOS 6 64-bit
- CentOS 7 64-bit
- Ubuntu 12.04 LTS 64-bit
- Ubuntu 14.04 LTS 64-bit
- Ubuntu 16.04 LTS 64 bit

1. You should have the latest stable release of Ansible.

1. It requires a 64-bit support kernel and 20 GB hard disk.

1. It must have a minimum of 4 GB RAM.

Features of Ansible Tower



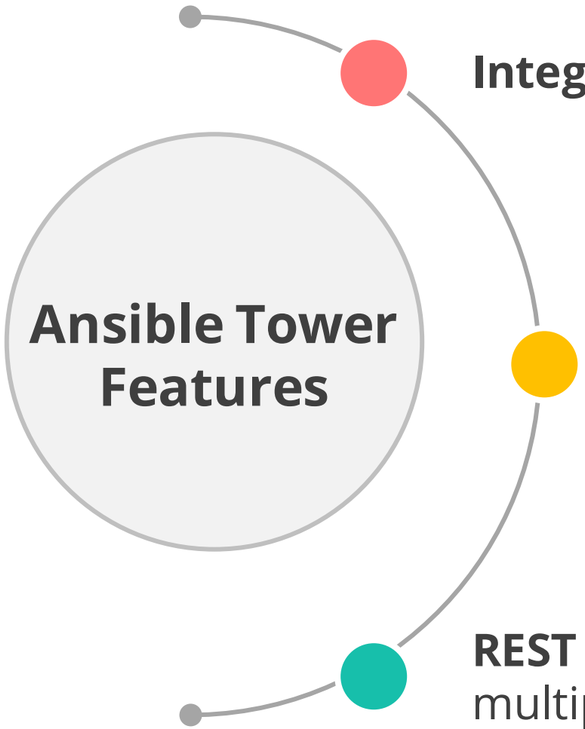
Dashboard: Displays everything which is going on in your Ansible environment

Real-Time Job Updates: Provides real-time updates and status of all the jobs or plays running in the environment

Cluster: Connects multiple Ansible Tower nodes into an Ansible Tower cluster which makes it scalable

Track Inventory: Manages your entire infrastructure by pulling inventory from public cloud providers

Features of Ansible Tower



Integrated Notification: Sets email notifications of job completion or failure

Schedule Ansible Jobs: Schedules jobs such as playbook runs, cloud inventory updates, and source control updates to run according to the need

REST API and Tower CLI Tool: Consists of an API to integrate tower with multiple environment

Key Takeaways

- Ansible module is the smallest piece of code or command that runs on the node or client machine.
- Ansible loops include changing ownership on several files and/or directories with the file module, creating multiple users, and repeating a step.
- Ansible conditionals are the control statements used in a playbook in order to generate accurate report or output.
- Ansible Role is a framework that applies certain collection of variables, tasks, files, templates, and modules based on a directory structure.