Caltech | Center for Technology & Management Education

Post Graduate Program in DevOps

simplilearn

Ansible Basics

Learning Objectives

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

- Explain the core components of Ansible
- Implement inventories
- Illustrate YAML
- Implement playbooks and groups
- Define upstream, downstream, and tags

Introduction

Basics of Ansible

Ansible is an open source IT engine used to automate application deployment, service orchestration, cloud services, and other IT tools.

ANSIBLE

Basics of Ansible

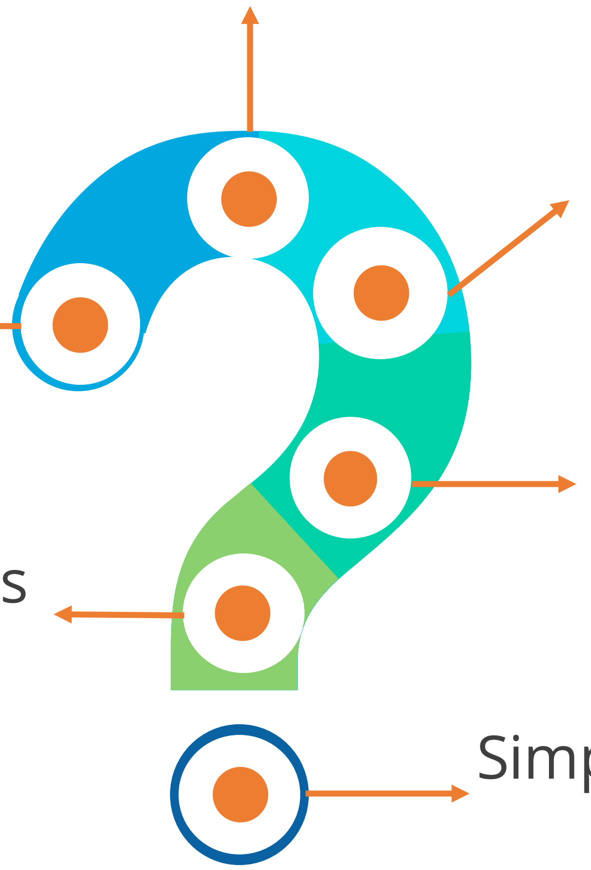
- Ansible uses playbook to describe automation jobs which are written in YAML.
- YAML is a human-readable data serialization language which is mainly used for configuration files.
- Ansible is designed for multi-tier deployment.
- Ansible models IT infrastructure by interrelating all the systems.

Basics of Ansible

- Ansible is completely agentless and works by connecting nodes primarily through SSH.
- Ansible pushes small programs, called Ansible modules, on the nodes and removes them when finished.
- Ansible manages inventory in simple text files called hosts file.
- Ansible uses the hosts file to control the actions on a specific group in the playbooks.

Why Use Ansible?

Below are the operational advantages of Ansible:



Lower system maintenance cost

Easy to structure release process with distributed and continuous development

Easy to audit and verify processes

Faster recovery time

Compatible with all infrastructures

Simple integration with cloud platforms

Why Use Ansible?

Below are the technical advantages of Ansible:

Agentless:

No additional software required to install Ansible

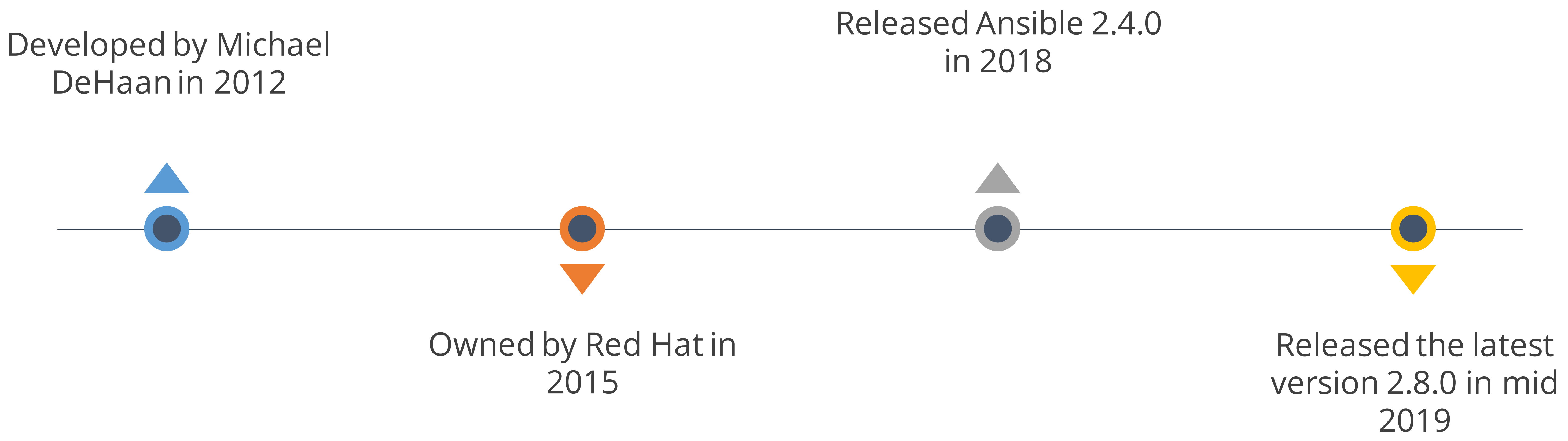
Flexible:

Multiple modules to manage systems, networks, workloads, and services

Efficient:

Playbooks, modules, facts, inventory, and other tools make it compatible with most of the infrastructures

History of Ansible



Developed by Michael DeHaan in 2012

Owned by Red Hat in 2015

Released Ansible 2.4.0 in 2018

Released the latest version 2.8.0 in mid 2019

Ansible Components

Control Machine
Machine on which Ansible is installed and acts as a server

Inventory
Hostfile that contains the information about the managed nodes

Playbook
Entry point of Ansible provisioning written in YAML

Task
Block of code that defines a single process

Ansible Components

Module
Abstract of a system task like creating and changing files

Role
Framework that organizes playbooks and other files to facilitate sharing and reuse provisioning

Facts
Global variables containing information about the system

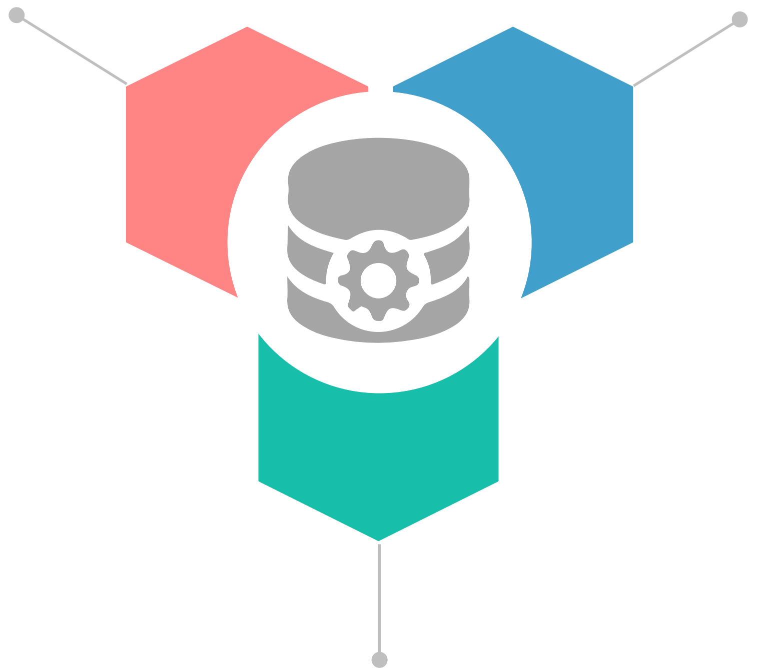
Handlers
Tasks that trigger changes in service status

Environment Setup

Prerequisites

Python 2.7 or higher

Minimum 8GB RAM



SSH or SCP communicator

Installing Ansible

Below are the different commands used to install Ansible on different operating systems:

- On Fedora:

$ sudo dnf install ansible

- On RHEL and CentOS:

$ sudo yum install ansible

- To enable the Ansible Engine repository for RHEL 8, run the following command:

$ sudo subscription-manager repos --enable ansible-2.9-for-rhel-8-x86_64-rpms

- To enable the Ansible Engine repository for RHEL 7, run the following command:

$ sudo subscription-manager repos --enable rhel-7-server-ansible-2.9-rpms

Installing Ansible

- To configure the PPA on your machine and to install Ansible on Ubuntu, run the below commands:

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

- For Debian, add the following line to the file **/etc/apt/sources.list**:

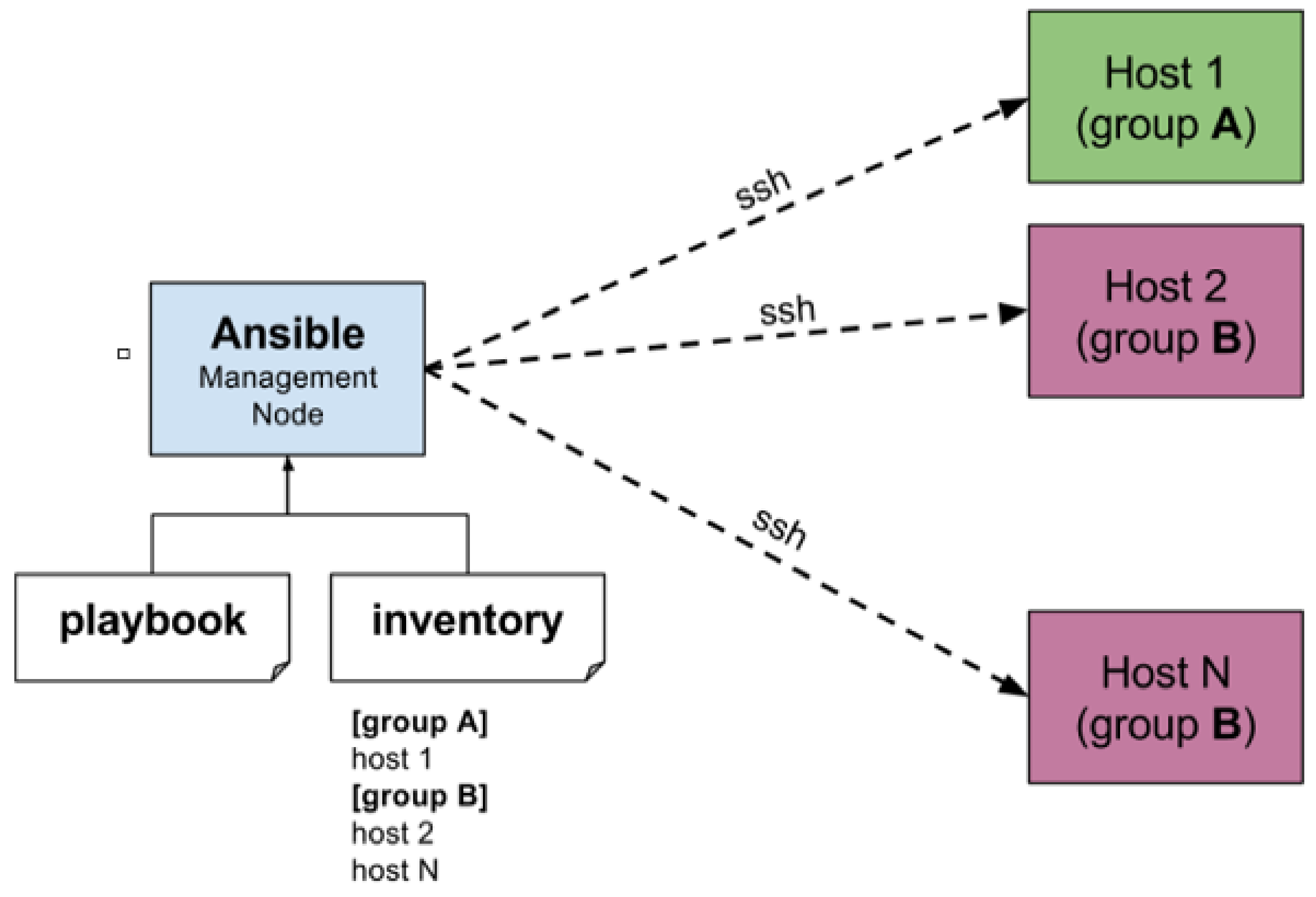
```
deb http://ppa.launchpad.net/ansible/ansible/ubuntu trusty main
```

- On completion, run the below commands:

```
$ sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
93C4A3FD7BB9C367
$ sudo apt update
$ sudo apt install ansible
```

Working of Ansible

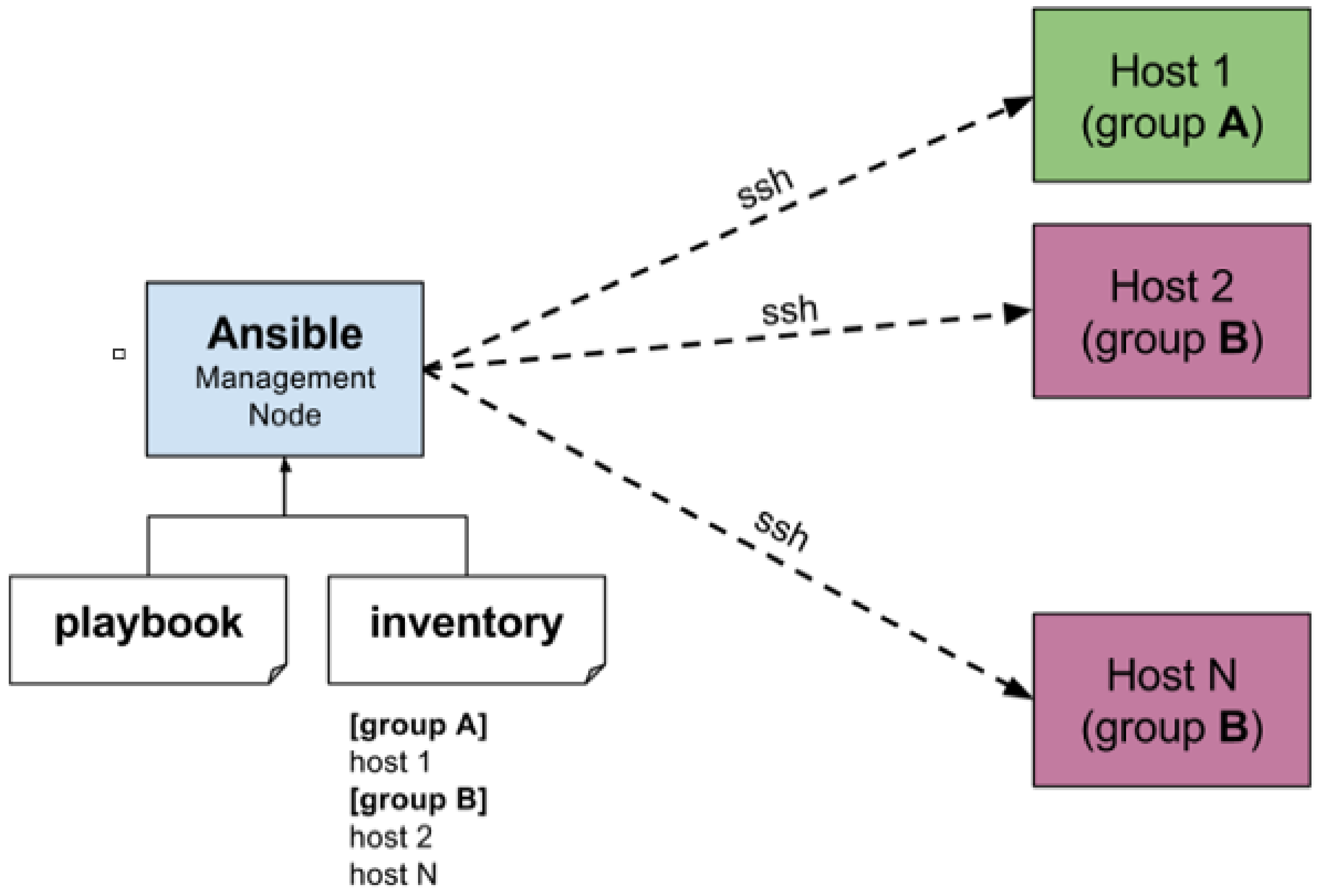
- Ansible works by connecting to nodes and pushing out small programs called Ansible modules.
- Ansible runs modules over SSH by default and removes them when finished.
- Modules can be stored on any machine without any servers, daemons, or databases.



Source: https://www.tutorialspoint.com/ansible/ansible_introduction.htm

Working of Ansible

- The management node is the controlling node.
- It controls the execution of the playbook which are the YAML code written to execute small tasks over the client machines.
- The inventory file is the list of hosts where the Ansible modules will run.
- Management node performs an SSH connection and runs modules on the hosts.



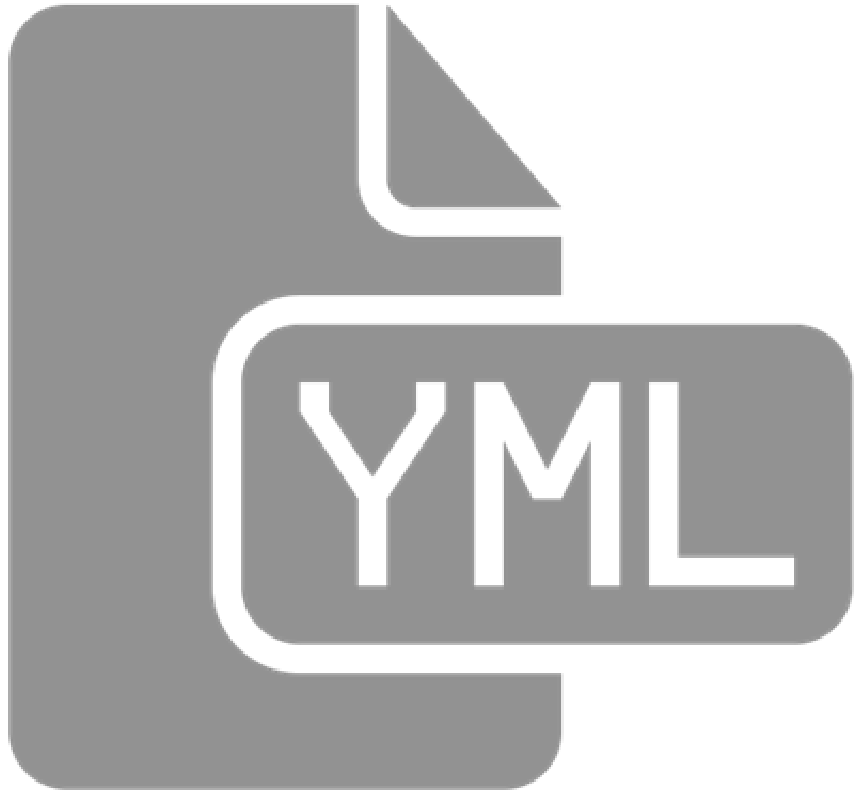
Source: https://www.tutorialspoint.com/ansible/ansible_introduction.htm

Introduction to YAML

Ansible uses YAML syntax for expressing Ansible playbooks.

- Every **YAML** file optionally starts with “---” and ends with “...”
- YAML uses simple key-value pair to represent the data.

```
--- #Optional YAML start syntax
james:
  name: james john
  rollNo: 34
  div: B
  sex: male
... #Optional YAML end syntax
```



Introduction to YAML

- Abbreviation can be used to represent dictionaries.

Example:
James: {name: james john, rollNo: 34, div: B, sex: male}

- Abbreviation can also be used to represent lists.

Example:
Countries: [‘America’, ‘China’, ‘Canada’, ‘Iceland’]

Introduction to YAML

- All members of a list are lines beginning at the same indentation level starting with a "- " (a dash and a space):

```
---
# This is a comment
- Apple
- Orange
- Strawberry
- Mango
...
```

- More complicated data structures are possible, such as lists of dictionaries, dictionaries whose values are lists or a mix of both:

```
# Employee records
-  martin:
name: Martin D'vloper
job: Developer skills:
-  python
-  perl
-  pascal

-  tabitha:
name: Tabitha Bitumen
job: Developer skills:
-  lisp
-  fortran
-  erlang
```

Introduction to YAML

- When you are creating a file in YAML, you should remember the following basic rules:

1. YAML is case sensitive.

1. The files should have **.yaml** as the extension.

1. YAML does not allow the use of tabs while creating YAML files; instead allows spaces.

Introduction to YAML

- Things to remember for YAML basic elements are given below:

1. Comments in YAML begin with the (**#**) character.

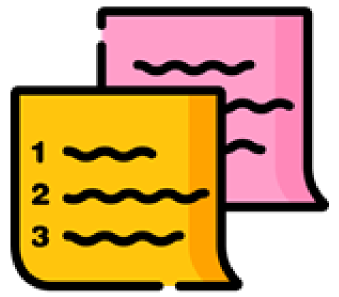
1. Comments must be separated from other tokens by whitespaces.

1. Indentation of whitespace is used to denote the structure.

1. Tabs are not included as indentation for YAML files.

1. List members are denoted by a leading hyphen (**-**)

1. List members are enclosed in square brackets and separated by commas.

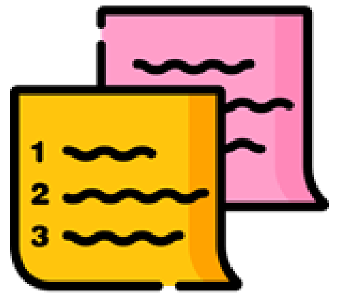


Source: https://www.tutorialspoint.com/yaml/yaml_basics.htm

Introduction to YAML

- Things to remember for YAML basic elements are given below:

1. Associative arrays are represented using colon **(:)** and enclosed in curly braces **{}**.
1. Multiple documents with single streams are separated by 3 hyphens (---).
1. Repeated nodes in each file are denoted by an ampersand (**&**) and an asterisk (*****).
1. Colons and commas are used as list separators followed by a space with scalar values.
1. Nodes are labeled with an exclamation mark (**!**) or double exclamation marks (**!!**).



Source: https://www.tutorialspoint.com/yaml/yaml_basics.htm

Ansible Ad-hoc Commands and Inventories

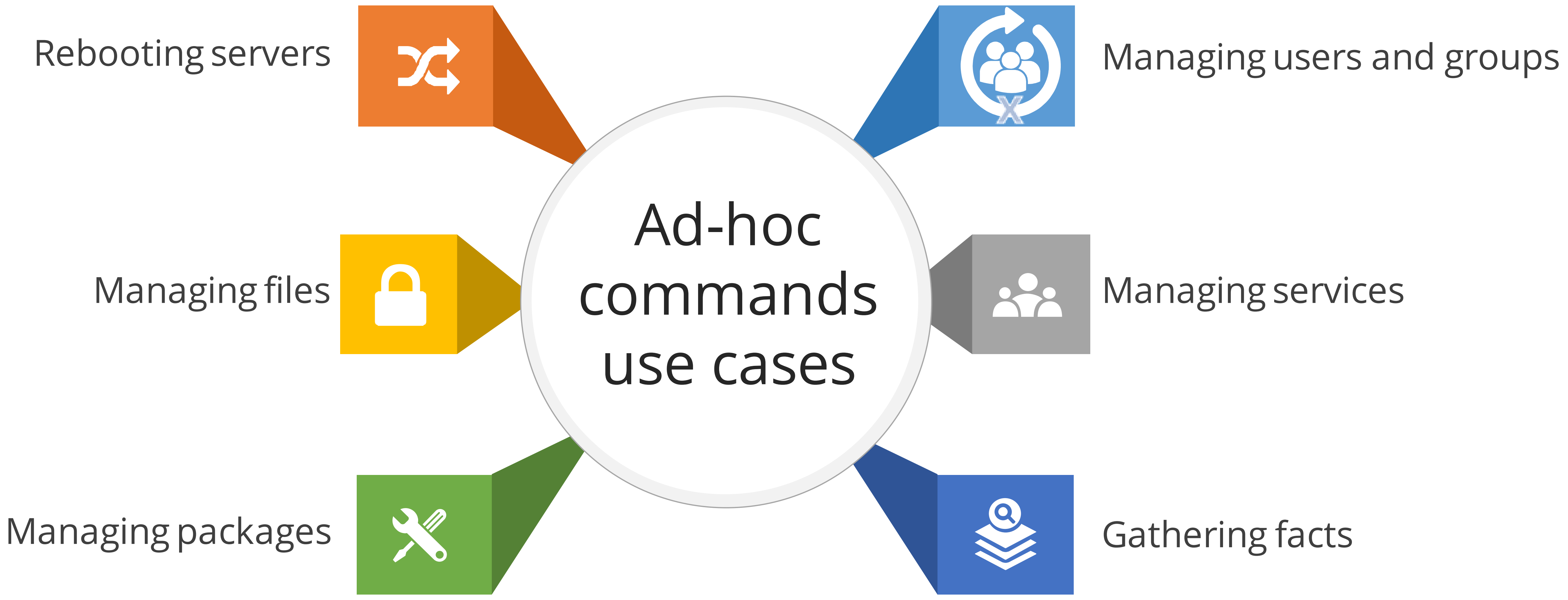
Ad-hoc Commands

Ansible ad-hoc command uses the **/usr/bin/ansible** command-line tool to automate a single task on one or more client nodes.

- Ad-hoc commands are great for tasks repeated rarely.
- For example, if you want to send greetings to the user on his or her birthday, you could execute a quick one-liner in Ansible without writing a playbook.
- An ad-hoc command looks like this:

```
$ ansible [pattern] -m [module] -a "[module options]"
```

Use Cases for Ad-hoc



Use Cases for Ad-hoc

- Rebooting servers
- Managing files
- Managing packages
- Managing users
- Managing services
- Gathering facts

- To reboot all the servers in a group:

```
$ ansible <groupname> -a "/sbin/reboot"
```

- To reboot the servers with multiple parallel forks:

```
$ ansible <groupname> -a "/sbin/reboot" -f 10
```

- To connect as a different user:

```
$ ansible <groupname> -a "/sbin/reboot" -f 10 -u username
```

- For privilege escalation, connect to the server with username and run the below command as the root user by using the **become** keyword:

```
$ ansible <groupname> -a "/sbin/reboot" -f 10 -u username --become [--ask-become-pass]
```

Use Cases for Ad-hoc

- Rebooting servers
- **Managing files**
- Managing packages
- Managing users
- Managing services
- Gathering facts

- To transfer a file directly to all servers in a group:

```
$ ansible <groupname> -m copy -a "src=/etc/hosts dest=/tmp/hosts"
```

- The file module allows changing ownership and permissions on files. These same options can be passed directly to the copy module as well:

```
$ ansible webservers -m file -a "dest=/srv/foo/a.txt mode=600"
$ ansible webservers -m file -a "dest=/srv/foo/b.txt mode=600 owner=mdehaan
group=mdehaan"
```

- The file module can also create directories, similar to mkdir -p:

```
$ ansible webservers -m file -a "dest=/path/to/c mode=755 owner=mdehaan group=mdehaan
state=directory"
```

- To delete directories (recursively) and files:

```
$ ansible webservers -m file -a "dest=/path/to/c state=absent"
```

Use Cases for Ad-hoc

- Rebooting servers
- Managing files
- **Managing packages**
- Managing users
- Managing services
- Gathering facts

- Use **yum** to install, update, or remove packages from nodes.
- To ensure a package is installed without updating it:

```
$ ansible webservers -m yum -a "name=acme state=present"
```

- To ensure a specific version of a package is installed:

```
$ ansible webservers -m yum -a "name=acme-1.5 state=present"
```

- To ensure a package is of the latest version:

```
$ ansible webservers -m yum -a "name=acme state=latest"
```

- To ensure a package is not installed:

```
$ ansible webservers -m yum -a "name=acme state=absent"
```

Use Cases for Ad-hoc

- Rebooting servers
- Managing files
- Managing packages
- **Managing users**
- Managing services
- Gathering facts

- To create, manage, and remove user accounts on your managed nodes with ad-hoc tasks:

```
$ ansible all -m user -a "name=foo password=<crypted password here>" $ ansible all -m
user -a "name=foo state=absent‟
```

Use Cases for Ad-hoc

- Rebooting servers
- Managing files
- Managing packages
- Managing users
- **Managing services**
- Gathering facts

- To ensure a service has started on all web servers:

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

- To restart a service on all web servers:

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

- To ensure a service is stopped:

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

Use Cases for Ad-hoc

Rebooting servers

Managing files

Managing packages

Managing users

Managing services

Gathering facts

- To see all facts:

```
$ ansible all -m setup
```

Inventory Basics

Inventory file contains the list of the managed nodes. Sometimes, it is also called the **hostfile.** It also organizes managed nodes, creating and nesting groups for scaling.

- Format of the inventory file depends on the plugin present in the system.
- The most common formats are INI and YAML.
- Below is a sample of an **INI etc/ansible/hosts** file:

```
mail.example.com

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

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

Inventory Basics

Below is the difference between INI and YAML inventory files:

INI format inventory file:

```
mail.example.com

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

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

YAML format inventory file:

```
all:
      hosts:
            mail.example.com:
      children:
            webservers:
                  hosts:
            foo.example.com:
            bar.example.com:
            dbservers:
                  hosts:
            one.example.com:
            two.example.com:
            three.example.com:
```

Inventory Groups

Let us understand the inventory file:

INI format inventory file:

```
mail.example.com: 9905

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

[dbservers]
One[1:50].example.com
two.example.com
three.example.com
```

- The heading in the brackets are group names.
- It decides at what time the policies are controlled.
- You can also put a node in more than one group.
- If the host runs on a non-standard SSH port, then specify the port number with a colon as shown in the first statement.
- You can also provide range to the hosts in brackets.

Inventory Groups

Here is an example of YAML file with nested groups:

- **one.example.com** is present in dbservers, east, and prod groups.

```
all:
  hosts:
    mail.example.com:
  children:
    webservers:
      hosts:
        foo.example.com:
        bar.example.com:
    dbservers:
      hosts:
        one.example.com:
        two.example.com:
        three.example.com:
    east:
      hosts:
        foo.example.com:
        one.example.com:
        two.example.com:
    west:
      hosts:
        bar.example.com:
        three.example.com:
    prod:
      hosts:
        foo.example.com:
        one.example.com:
        two.example.com:
    test:
      hosts:
        bar.example.com:
        three.example.com:
```

Inventory Variables

Below are the different places where you can assign variables to the hosts that will be used in the playbooks.

Hosts Variables

- You can assign the variables to the hosts that will be used in playbooks, as shown below:

```
[atlanta]
host1 http_port=80 maxRequestsPerChild=808
host2 http_port=303 maxRequestsPerChild=909
```

Inventory Variables

Group Variables

- Apply variables to an entire group at once as shown below:

```
[atlanta]
host1
host2

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

Inventory Variables

Groups of Groups and Group Variables

- It is possible to make groups of the group using the **:children's** suffix. You can apply variables using **:vars**.

```
[atlanta]
host1
host2

[raleigh]
host2
host3

[southeast: children]
Atlanta
Raleigh

[southeast:vars]
some_server=foo.southeast.example.com
halon_system_timeout=30
self_destruct_countdown=60
escape_pods=2

[usa: children]
southeast
northeast
southwest
northwest
```

Source: https://www.javatpoint.com/ansible-inventory

Inventory Variables

Assigning a variable to a group:
If all hosts in a group share a variable value, you can apply that variable to an entire group at once.

INI

```
[atlanta]
host1
host2

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

YAML

```
atlanta:
  hosts:
    host1:
    host2:
  vars:
    ntp_server:
ntp.atlanta.example.com
    proxy: proxy.atlanta.example.com
```

Inventory Variables

Inheriting variable values: group variables for groups of groups
To make groups of groups use the **:children** suffix in INI or the **children:** entry in YAML.
To apply variables to these groups of groups **using :vars or vars:**

INI

```
[atlanta]
host1
Host2

[raleigh]
host2
host3

[southeast:children]
atlanta
raleigh

[southeast:vars]
some_server=foo.southeast.example.com
halon_system_timeout=30
self_destruct_countdown=60
escape_pods=2

[usa:children]
southeast
northeast
southwest
northwest
```

YAML

```
all:
  children:
    usa:
      children:
        southeast:
          children:
            atlanta:
              hosts:
                host1:
                host2:
            raleigh:
              hosts:
                host2:
                host3:
          vars:
            some_server: foo.southeast.example.com
            halon_system_timeout: 30
            self_destruct_countdown: 60
            escape_pods: 2
        northeast:
        northwest:
        southwest:
```

Inventory Variables

Properties of child groups:

1. A child member is automatically a member of the parent group.

1. A child's variable will have higher precedence than a parent's variable.

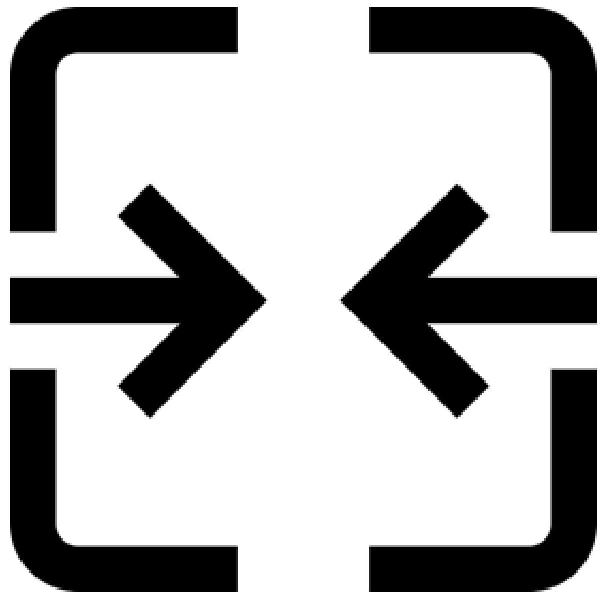
1. Groups can have multiple parents and children, but it cannot be bidirectional.

1. If a host is present in multiple groups, only one instance of a host can gather data from other instances of the same host.

Inventory Variables

Merging Variables:

- Variables are merged with a specific host before a play is run.
- Ansible overwrites variables defined for a group or host.
- The precedence of merging variables is:
    - all group (It is the parent of other groups.)
    - parent group
    - child group
    - host
- Ansible merges groups at the same parent-child level alphabetically.
- The last group overwrites the previously loaded groups.



Dynamic Inventory

- Ansible can generate host inventory dynamically.
- Dynamic inventory provides information about public and private cloud providers, cobbler system information, LDAP database, and CMDB (Configuration Management database).
- For cloud providers, authentication and access information should be defined in files that the script can access.
- Dynamic inventory program can be written in any programming language and must return in JSON format.

Dynamic Inventory

Ansible uses the inventory script to retrieve host information from an external inventory system.
This script should support the **–list** parameter, returning host group and hosts information.

Example of a dynamic inventory:

```
[linuxtechi@control-node test-lab]$ ./inventoryscript --
list
{
  "webservers" :["web1.example.com", "web2.example.com"
],
  "dbservers" :["db1.example.com", "db2.example.com"]
}
```

Assisted Practice

Apache Server Set Up with Ansible

Problem Statement:

You are given a project to set up apache server using inventory and adhoc commands.

Assisted Practice: Guidelines

Steps to perform:

1. Install Ansible
2. Establish connectivity between controller and node machine
3. Write Ansible YAML script to install ansible software
4. Execute Ansible YAML script

Ansible Playbooks

Introduction to Playbooks

Playbooks are an ordered list of tasks saved in a file, which can be used to repeatedly run those tasks.

- Playbooks are written in YAML and are easy to read, write, share, and understand.
- Each playbook contains one or more **plays** in a list.
- A **play** is responsible in mapping the hosts to well-defined roles represented by ansible tasks.
- **Play** can also be used to orchestrate multiple machine deployment and running processes on target machines.

Introduction to Playbooks

Example of a playbook **verify-apache.yml** that contains just one play:

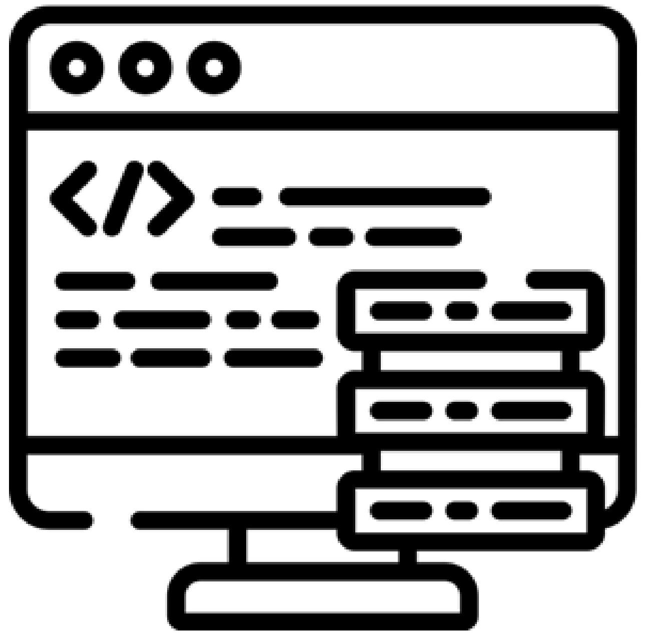
```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf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

Components of Playbooks

Hosts and Users:

- For each play in a playbook, a target machine is selected on which the tasks are executed.
- The host line is a list of one or more groups or host patterns separated by colons.
- The **remote_user** command refers to the name of the user account.

```
---
- hosts: webservers
  remote_user: root
```



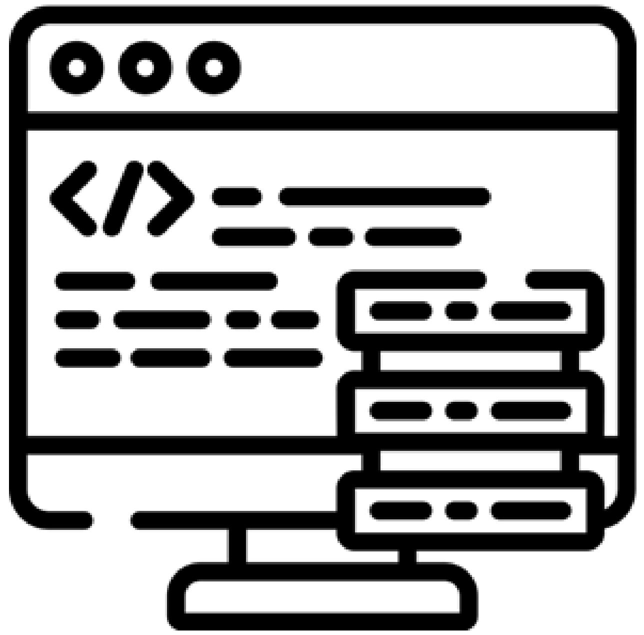
Components of Playbooks

- Remote users can also be defined per task as shown below:

```
---
- hosts: webservers
  remote_user: root
  tasks:
    - name: test connection
      ping:
      remote_user: yourname
```

- Use the keyword **become** on a particular task instead of the whole play to change the user account.

```
---
- hosts: webservers
  remote_user: yourname
  become: yes
```



Components of Playbooks

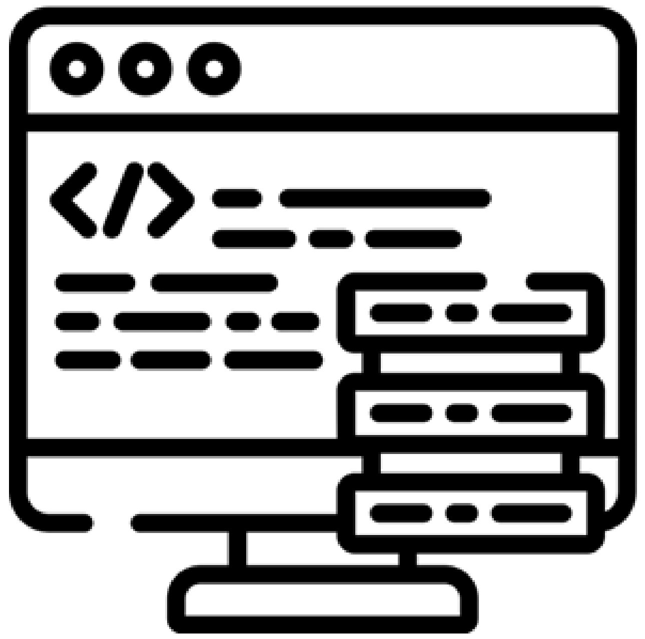
Plays and Tasks:

- Each play contains a list of tasks.
- Task is nothing but small operations that are executed on the target machine.
- Tasks are executed one at a time in an order.
- Within a play, all hosts get the same task directives.
- A play maps the hosts to the corresponding tasks.

Components of Playbooks

Plays and Tasks:

- During playbook execution, hosts with failed tasks are taken out of the rotation for the entire playbook.
- The goal of a task is to execute a module with very specific arguments.
- To achieve short execution time, modules should be idempotent.
- It is recommended to check the module's state if its final state has been achieved and the execution can be stopped if it's true.
- Rerunning of the plays becomes idempotent if the modules are idempotent.



Components of Playbooks

Ansible Handlers:

- Handlers are the **notify** actions that are triggered at the end of each block of tasks in a play.
- It is only triggered once.
- Below is an example of restarting two services when the contents of a file change. The operations present in the notify section are called handlers.

```
- name: template configuration file
  template:
    src: template.j2
    dest: /etc/foo.conf
  notify:
     - restart memcached
     - restart apache
```

Components of Playbooks

Handlers can also use **listen** keyword to trigger the generic tasks as shown below:

```
handlers:
    - name: restart memcached
      service:
        name: memcached
        state: restarted
      listen: "restart web services"
    - name: restart apache
      service:
        name: apache
        state: restarted
      listen: "restart web services"

tasks:
    - name: restart everything
      command: echo "this task will restart the web services"
      notify: "restart web services"
```

Components of Playbooks

- To run a playbook using a parallelism level of 10, use the command given below:

```
ansible-playbook playbook.yml -f 10
```

Linting Playbooks:

- **ansible-lint** runs a check on playbooks before executing them.
- For example:

```
$ ansible-lint verify-apache.yml
[403] Package installs should not use latest
verify-apache.yml:8
Task/Handler: ensure apache is at the latest version
```

Configure Error Handling

Ansible checks the return value of commands to detect runtime errors. The different ways to handle errors to continue the execution of play are:

- Ignoring failed commands
- Resetting unreachable hosts
- Using handlers and failures
- Controlling failure conditions

Configure Error Handling

- Overriding changed results
- Aborting plays
- Using blocks

Configure Error Handling

Ignoring failed commands:

- Playbooks stop executing steps on a host with a failed task.
- To ignore this and continue execution, write a task as shown below:

```
- name: this will not be counted as a failure
  command: /bin/false
  ignore_errors: yes
```

- This feature only works when the task should run but returns a **failed** value.

Configure Error Handling

Resetting unreachable hosts:

- Connection failures set hosts as **UNREACHABLE.**
- This removes the hosts from the list of active hosts for the run.
- **meta: clear_host_errors** can be used to reactivate all currently flagged hosts.

Using handlers and failures:

- When a task fails on a host, handlers which were previously notified are not allowed to run on that host.
- **--force-handlers** command-line option or **force_handlers: True** can be included in a play to explicitly trigger tasks.
- **force_handlers = True** can be added in ansible.cfg to perform the trigger action.

Configure Error Handling

Controlling failure conditions:

- User can define what "failure" means in each task using the **failed_when** conditional.
- Lists of multiple **failed_when** conditions are joined with an implicit action.
- The task only fails when all conditions are met.
- Define the conditions in a string with an operator to trigger a failure when any condition is true.

Configure Error Handling

- The two ways to check failure are:

1. 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"
```

2. 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
```

Configure Error Handling

- Follow the below task to combine multiple conditions for failure:

```
- name: Check if a file exists in temp and fail task if it does
  command: ls /tmp/this_should_not_be_here
  register: result
  failed_when:
    - result.rc == 0
    - '"No such" not in result.stdout'
```

- Follow the below task to fail the task when only one condition is satisfied:

```
failed_when: result.rc == 0 or "No such" not in result.stdout
```

- Follow the below task to split conditions into a multi-line yaml value with **>**:

```
- name: example of many failed_when conditions with OR
  shell: "./myBinary"
  register: ret
  failed_when: >
    ("No such file or directory" in ret.stdout) or
    (ret.stderr != '') or
    (ret.rc == 10)
```

Configure Error Handling

Overriding changed results:

- When a module runs it will typically report **changed** status based on whether it affected the machine state.
- For example:

```
tasks:

  - shell: /usr/bin/billybass --mode="take me to the river"
    register: bass_result
    changed_when: "bass_result.rc != 2"

  # this will never report 'changed' status
  - shell: wall 'beep'
    changed_when: False
```

- Multiple conditions can be combined to override **changed** status.

```
- command: /bin/fake_command
  register: result
  ignore_errors: True
  changed_when:
    - '"ERROR" in result.stderr'
    - result.rc == 2
```

Configure Error Handling

Aborting the play:

- There are cases when it is recommended to abort the entire play on failure.
- **any_errors_fatal** option will end the play and prevent any other plays from running.
- It can be set at the play or block level as shown below:

```
- hosts: somehosts
  any_errors_fatal: true
  roles:
    - myrole

- hosts: somehosts
  tasks:
    - block:
        - include_tasks: mytasks.yml
      any_errors_fatal: true
```

- **max_fail_percentage** can be used to abort the run after a given percentage of hosts have failed.

Configure Error Handling

Using blocks:

- Blocks level constraints handle errors similar to exceptions in most programming languages.
- Blocks only deal with **failed** status of a task.
- For example:

```
tasks:
- name: Handle the error
  block:
    - debug:
        msg: 'I execute normally'
    - name: i force a failure
      command: /bin/false
    - debug:
        msg: 'I never execute, due to the above task failing, :-('
  rescue:
    - debug:
        msg: 'I caught an error, can do stuff here to fix it, :-)`
```

Assisted Practice
Running First Playbook

Problem Statement:

You are given a project to install NodeJs using playbook.

Assisted Practice: Guidelines

Steps to perform:

1. Create a playbook
2. Run Ansible YAML script

Assisted Practice

MySql Set Up Using Ansible

Problem Statement:

You are given a project to set up MySql using playbook on a remote server.

Assisted Practice: Guidelines

Steps to perform:

1. Establish connectivity between controller and node machine
2. Write Ansible YAML script to install ansible software
3. Execute Ansible YAML script

Key Takeaways

- Ad-hoc commands are used for automatically running rarely repeated tasks.
- Ansible inventory is a file that contains information about the managed hosts.
- A playbook can have multiple plays that perform different tasks in the system.
- Each Ansible playbook works with an inventory file.