DevOps Certification Training

Configuration Management (CM) with Ansible

Learning Objectives

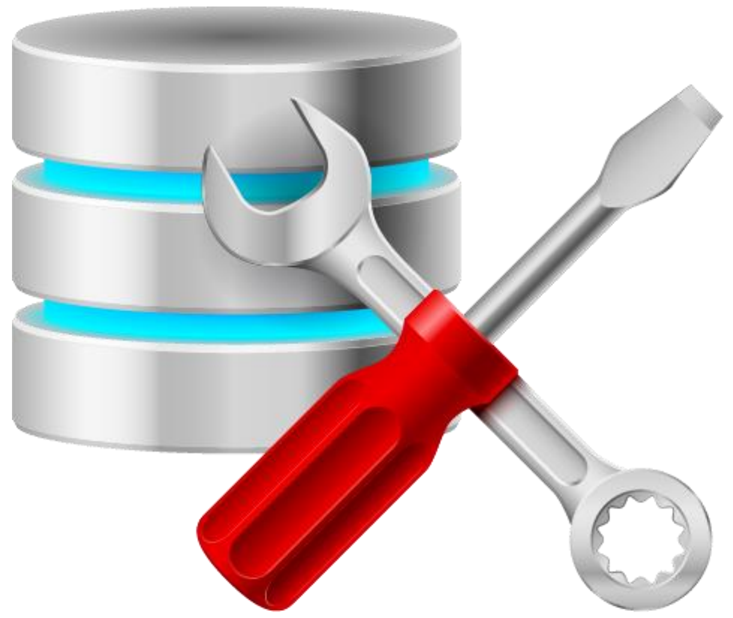
By the end of this lesson, you will be able to:

- Describe different configuration management tools
- Explain the Ansible architecture and terminology
- Demonstrate the creation of Ansible and working with it
- Illustrate YAML
- Explain different components of Terraform

Introduction to Configuration Management (CM)

Configuration Management

- Configuration management (CM) is a system engineering method that ensures a product's characteristics remain consistent during its life cycle.
- In the technology world, configuration management is an IT management mechanism that monitors individual configuration items of an IT system.



Features of Configuration Management

Enforcement
Prevents configuration drift

Concurrency Management
Manages concurrency properly

Version Control
Saves every change made to the file

Synchronization
Allows to check out more than one file

Benefits of Configuration Management

- It helps in increasing the efficiency with a well-defined configuration process that improves visibility and provides control with the help of tracking.
- It helps in cost optimization by having detailed knowledge of all the IT elements of the configuration, which helps to avoid unnecessary duplication.
- It provides greater agility and faster problem resolution, giving a better quality of service to the consumers.
- It enhances system and process reliability by detecting and correcting incorrect configurations before a detrimental effect on results.
- It also provides faster restoration of your service if a process failure occurs, i.e If the appropriate state of configuration will be known, restoring the working configuration will be much faster and easier.

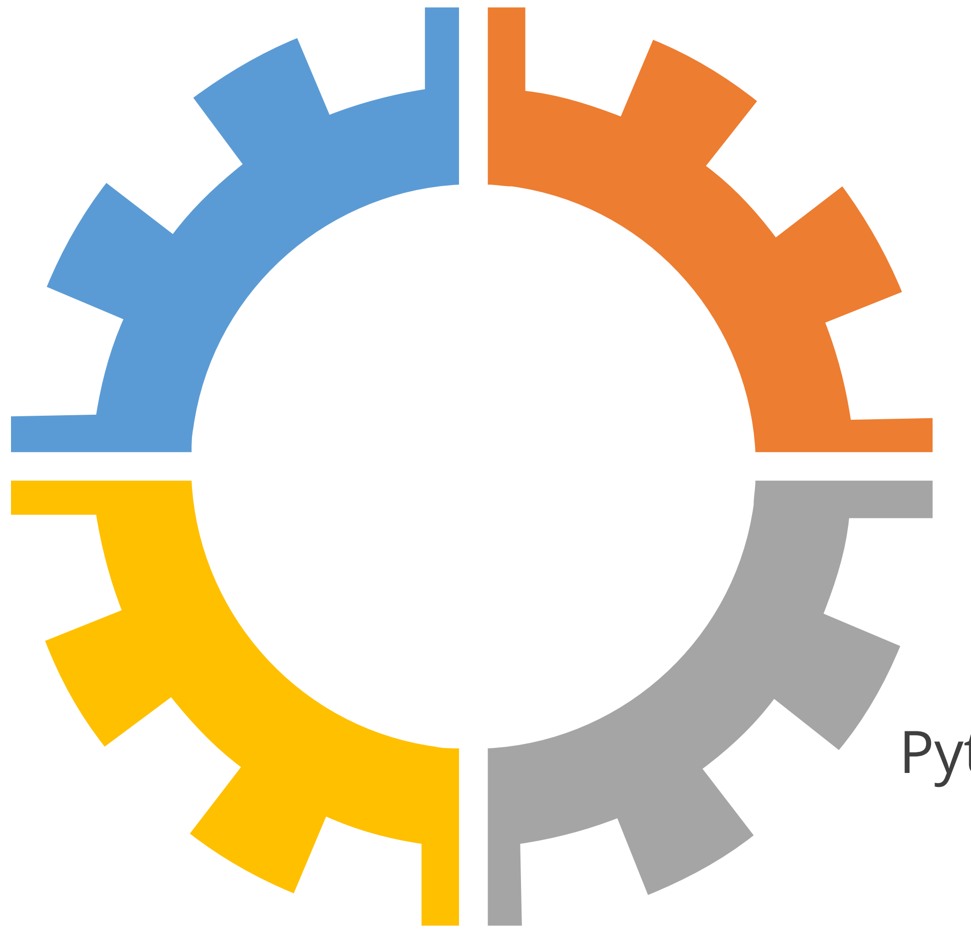
Role of Configuration Management in DevOps

DevOps is a concept that covers both the growth and operations phases of software development and so does configuration management.

Comprehensive configuration management is the term used in DevOps to describe configuration management, which is made up of:

- **Source Code Repository**: Mostly used during the development process
- **Artifact Repository**: Used during the production and operations phases of a project
- **Configuration Management Database**: Used during the production and operations phases of a project

Configuration Management Tools



Puppet
Ruby DSL-based CM tool used for managing software, systems, and network configuration items

Chef
Ruby-based CM tool having integration with most of the cloud-based platforms

Ansible
Python-based CM tool, also considered as agentless CM tool

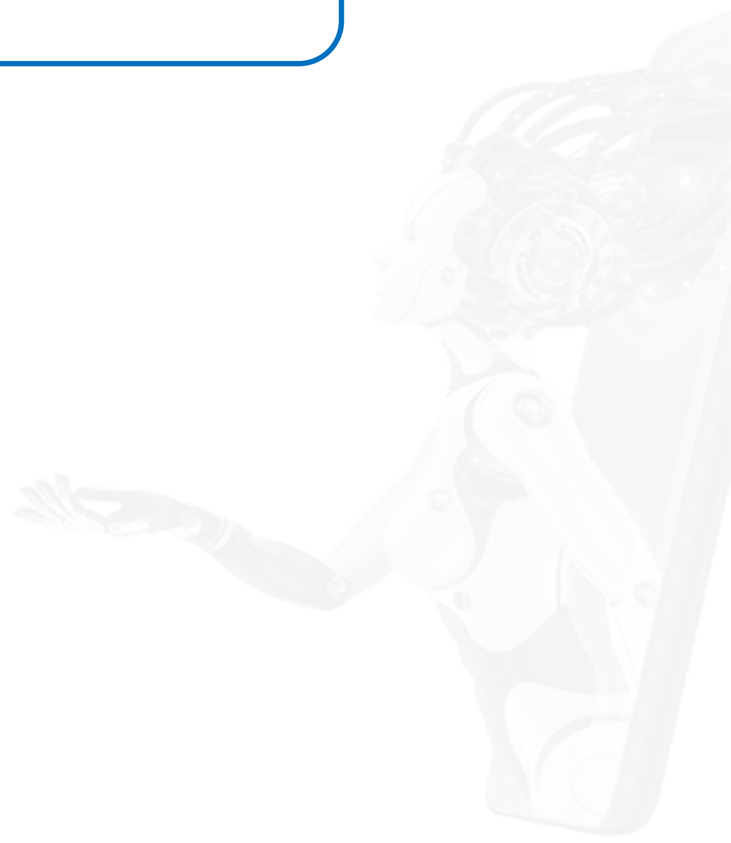
SaltStack
Python-based open-source CM tool used to remotely manage configuration items

TECHNOLOGY

Introduction to Ansible

Introduction to Ansible

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

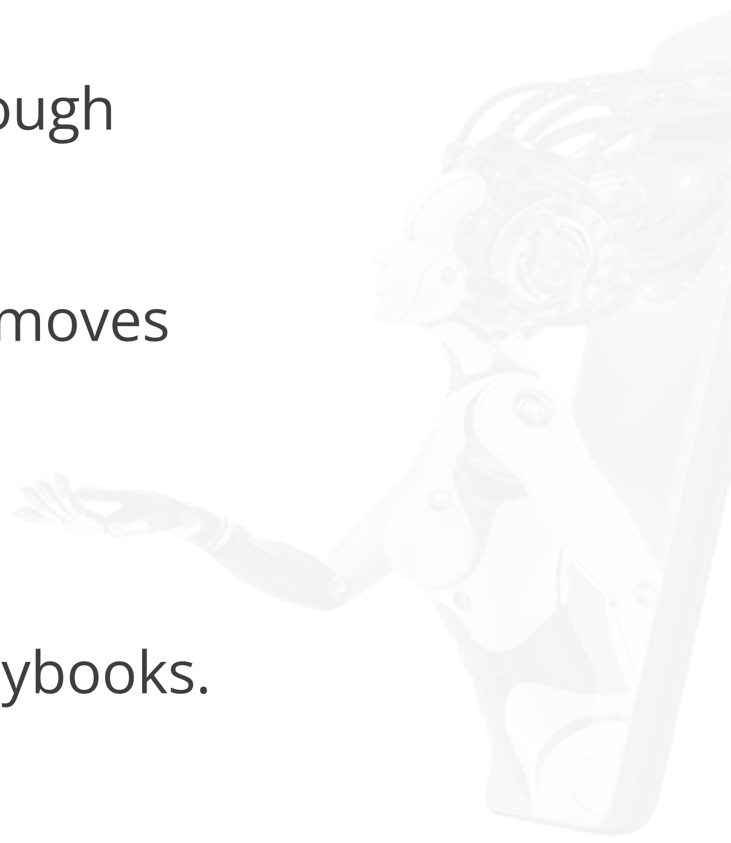


Basics of Ansible

- 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

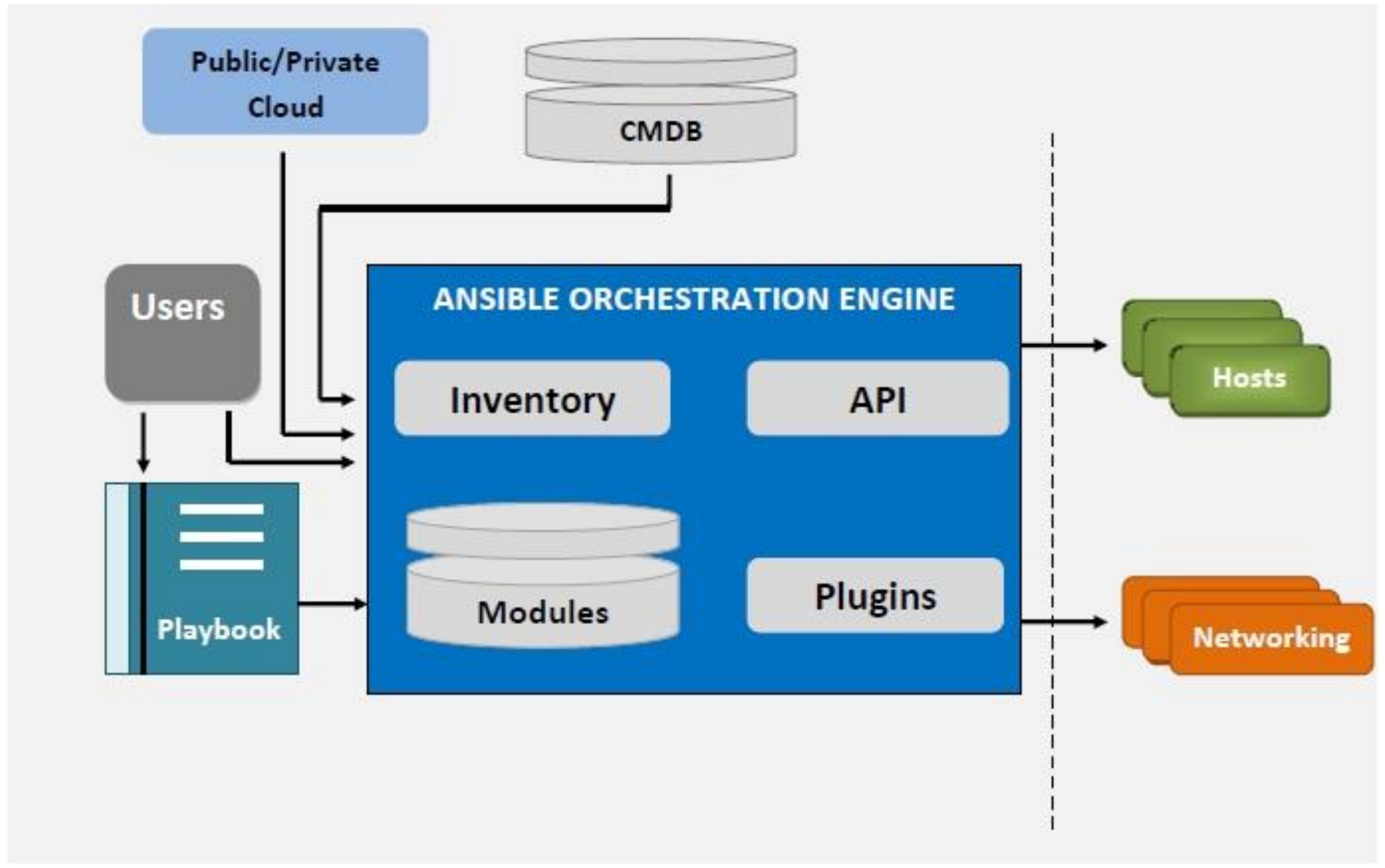
- Ansible is completely agentless and works by connecting nodes primarily through SSH.
- Ansible pushes small programs, called Ansible modules, on the nodes and removes them when finished.
- Ansible manages inventory in simple text files called hosts file.
- Ansible uses the hosts file to control the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



Ansible Architecture

- Ansible is an IT automation engine that automates cloud provisioning, configuration management, application deployment, intra-service orchestration, and a variety of other IT tasks.

- Ansible was built specifically for multi-tier implementations, and instead of handling one system at a time, it models the IT infrastructure by explaining how all of your systems interact.



source: devops.com

Ansible Architecture

Here's a brief description of how Ansible works and how the pieces fit together.

Modules

Module utilities

Plugins

Inventory

Playbooks

The Ansible search path

- Ansible connects to your nodes and sends scripts known as **Ansible modules**.
- You can write your own modules.
- The majority of modules accept parameters that define the system's desired state. Ansible then runs these modules (by default over SSH) and removes them when they're finished.
- Your module library can be stored on any computer, and no servers, daemons, or databases are needed.

Ansible Architecture

- Modules
- **Module utilities**
- Plugins
- Inventory
- Playbooks
- The Ansible search path

- Ansible stores function as module utilities when several modules use the same code, to reduce duplication and maintenance.
- Module utilities can be also created. However, only Python or PowerShell can be used to build module utilities.

Ansible Architecture

Modules

Module utilities

Plugins

Inventory

Playbooks

The Ansible search path

- Ansible's core functionality is augmented by plugins.
- Plugins execute on the control node within the /usr/bin/ansible method, while modules execute on the target system in separate processes (usually on a remote system).
- Ansible comes with many useful plugins, and can also be created..

Ansible Architecture

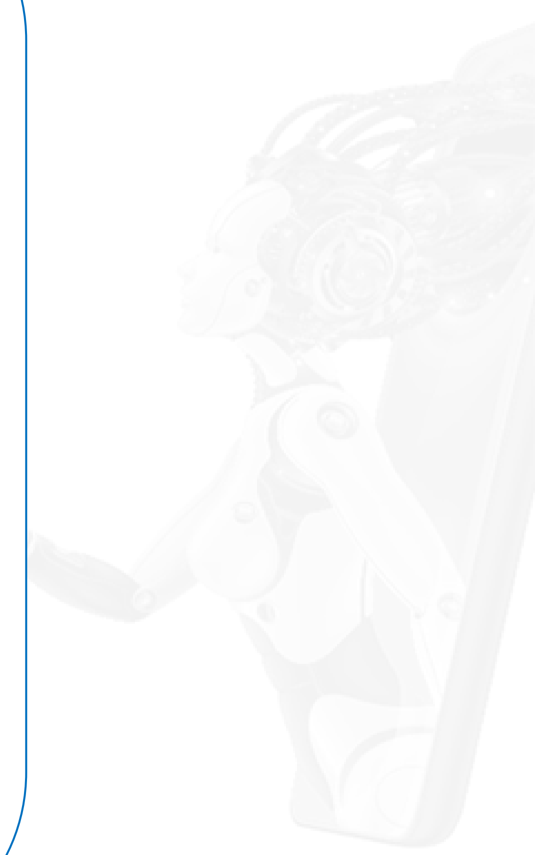
- Modules
- Module utilities
- Plugins
- **Inventory**
- Playbooks
- The Ansible search path

- The hosts and groups of hosts on which commands, modules, and tasks in a playbook run are described in the Ansible inventory file.
- Depending on Ansible setting and plugins, the file can be in a number of formats.
- /etc/ansible/hosts is the default location for the inventory register.
- Project-specific inventory files can also be created in different locations if necessary.

Ansible Architecture

- Modules
- Module utilities
- Plugins
- Inventory
- **Playbooks**
- The Ansible search path

- An Ansible playbook is a blueprint for automation tasks, which are complex IT tasks carried out with little to no human intervention.
- Ansible playbooks are simply frameworks, or pre-written code that developers can use as a starting point.
- IT infrastructure, networks, security systems, and developer personas are all routinely automated using Ansible playbooks.



Ansible Architecture

- Modules
- Module utilities
- Plugins
- Inventory
- Playbooks
- The Ansible search path

- Modules, module utilities, plugins, playbooks, and tasks can all be stored in different locations.
- Several files with similar or identical names in different locations can be available on the Ansible control node, if own code will be written to expand Ansible's core features.
- On any given playbook run, the search path decides which of these files Ansible can find and use.

Ansible Ad-Hoc Commands

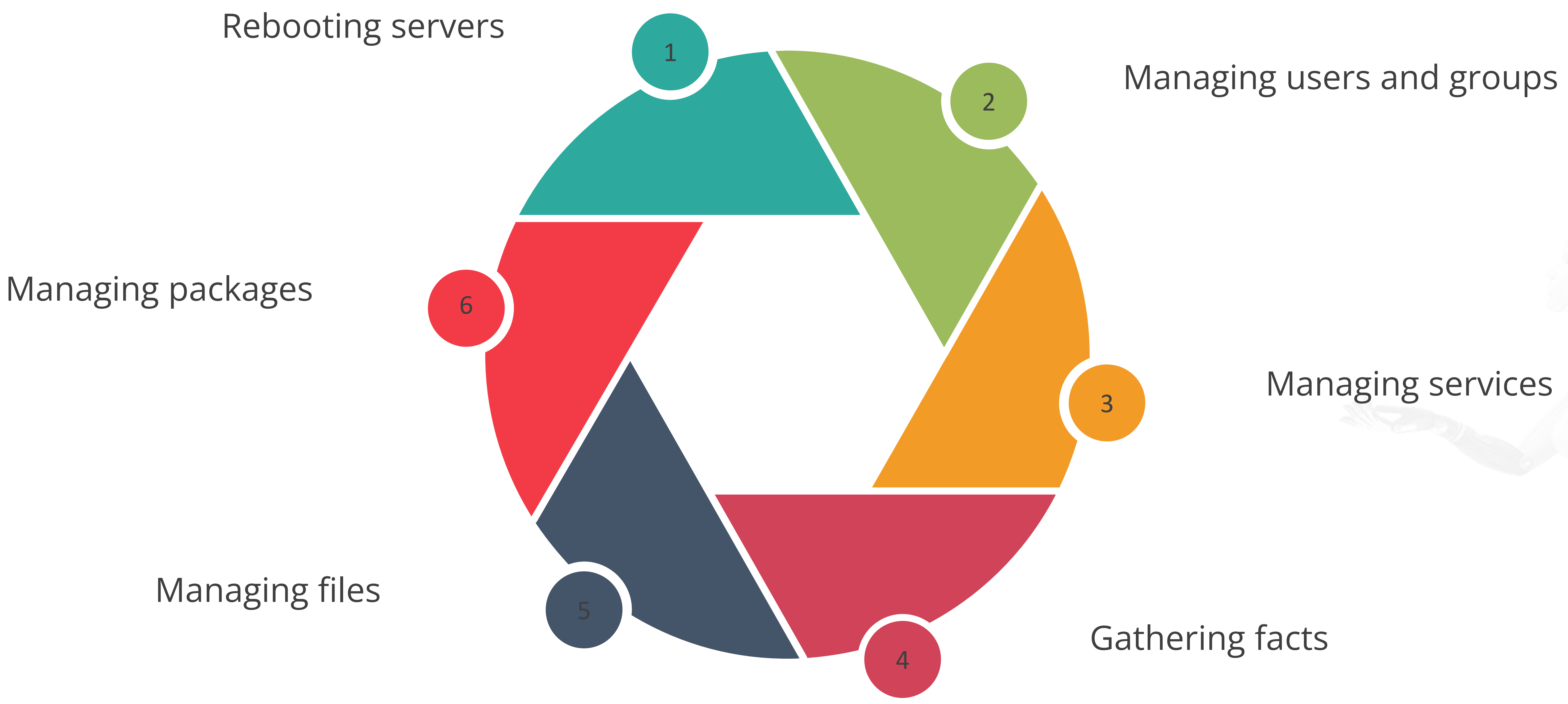
The /usr/bin/ansible command-line tool is used in an Ansible ad-hoc command to automate a single task on one or more controlled nodes.

Why use ad-hoc command?

- Ad-hoc commands are ideal for tasks that are only performed 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 Tasks



Use Cases for Ad-Hoc Tasks

- 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 Tasks

- 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 Tasks

- 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 Tasks

- 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 Tasks

- 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 Tasks

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

- To see all facts:

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

Seting Up Ansible

Duration: 15 Min.

Problem Statement: You are given a project to set up ansible in your system.

Prerequisites:

- Python 2.7 or higher
- Minimum 8 GB RAM
- SSH or SCP communicator

Unassisted Practice: Guidelines

Steps to set up Ansible:

1. Install Ansible on Ubuntu

YAML Scripting

Introduction to YAML

YAML syntax is simpler for humans to read and write than other popular data formats like XML or JSON. Hence, ansible uses it to express Ansible playbooks.

- Any YAML file begins with a list of items. Every item in the list is a key/value pair list, also known as a **hash** or **dictionary**.
- Every **YAML** file optionally starts with **---** and ends with **...**

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

YAML Scripting

The following basic rules should be kept in mind when creating a YAML file:

- YAML is case sensitive.
- The files should have **.yaml** as the extension.
- YAML does not allow the use of tabs while creating YAML files; instead allows spaces.

YAML Scripting

The following are some basic YAML elements to remember:

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.

YAML Scripting

The following are some basic YAML elements to remember:

7. Associative arrays are represented using colon : and enclosed in curly braces {}.

8. Multiple documents with single streams are separated by 3 hyphens ---.

9. Repeated nodes in each file are denoted by an ampersand & and an asterisk *.

10. Colons and commas are used as list separators followed by a space with scalar values.

11. Nodes are labeled with an exclamation mark ! or double exclamation marks !!.

TECHNOLOGY

Working with Ansible

Ansible as CM Tool

All in one: simplifies automation
All automation steps can be performed with Ansible due to the fact that Ansible is developed in python, making it easy to extend.

Lower learning curve
Ansible is simple to install and configure.

Mutable infrastructure
Ansible adapts well to mixed and automated environments.

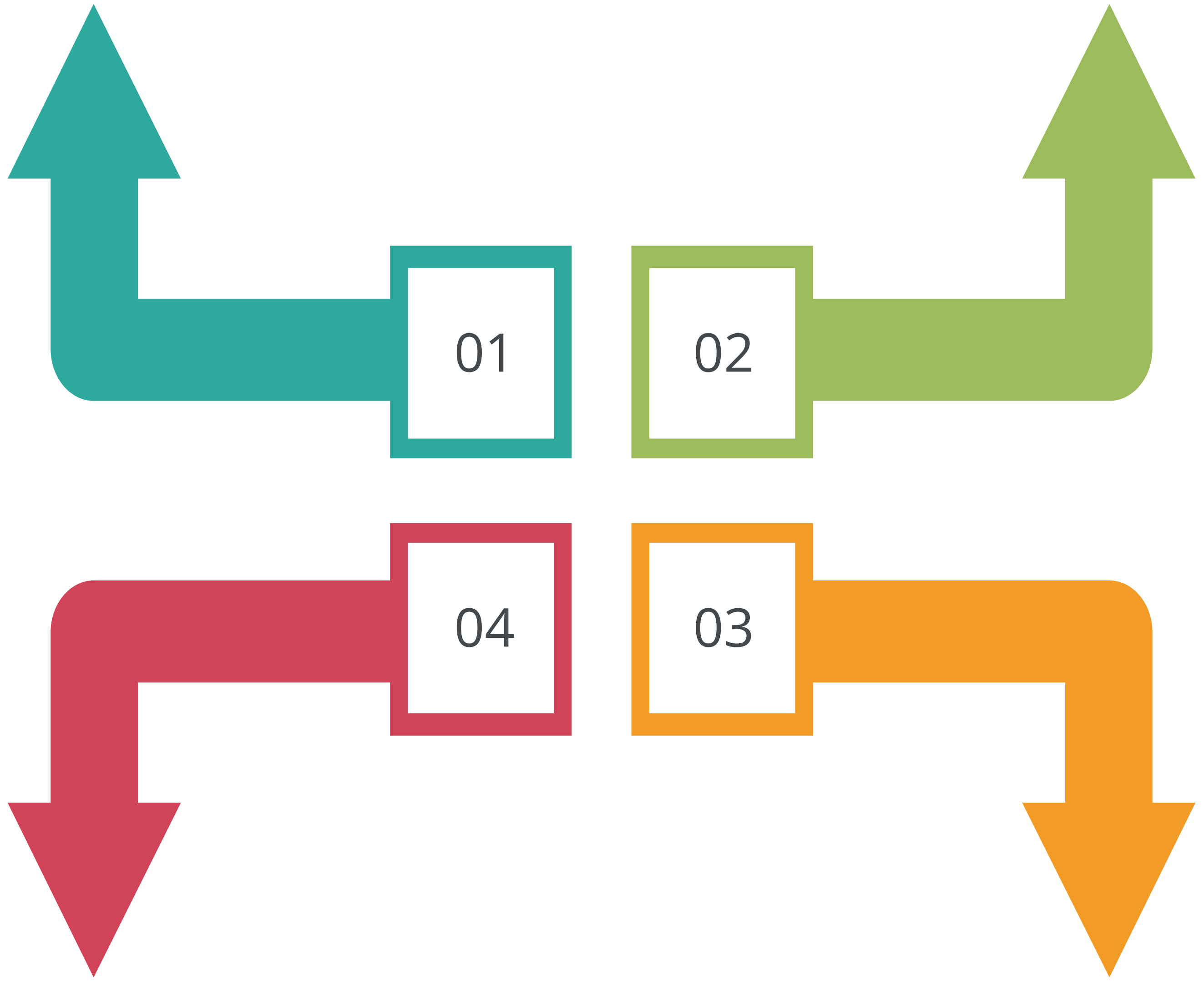
Agentless
Ansible would not necessitate the installation of agents on the endpoints.

Instant automation
Automation can be started as soon as the host is connected using Ansible.

Components of Ansible

Inventory
Host file that contains the information about the managed nodes

Task
Block of code that defines a single process



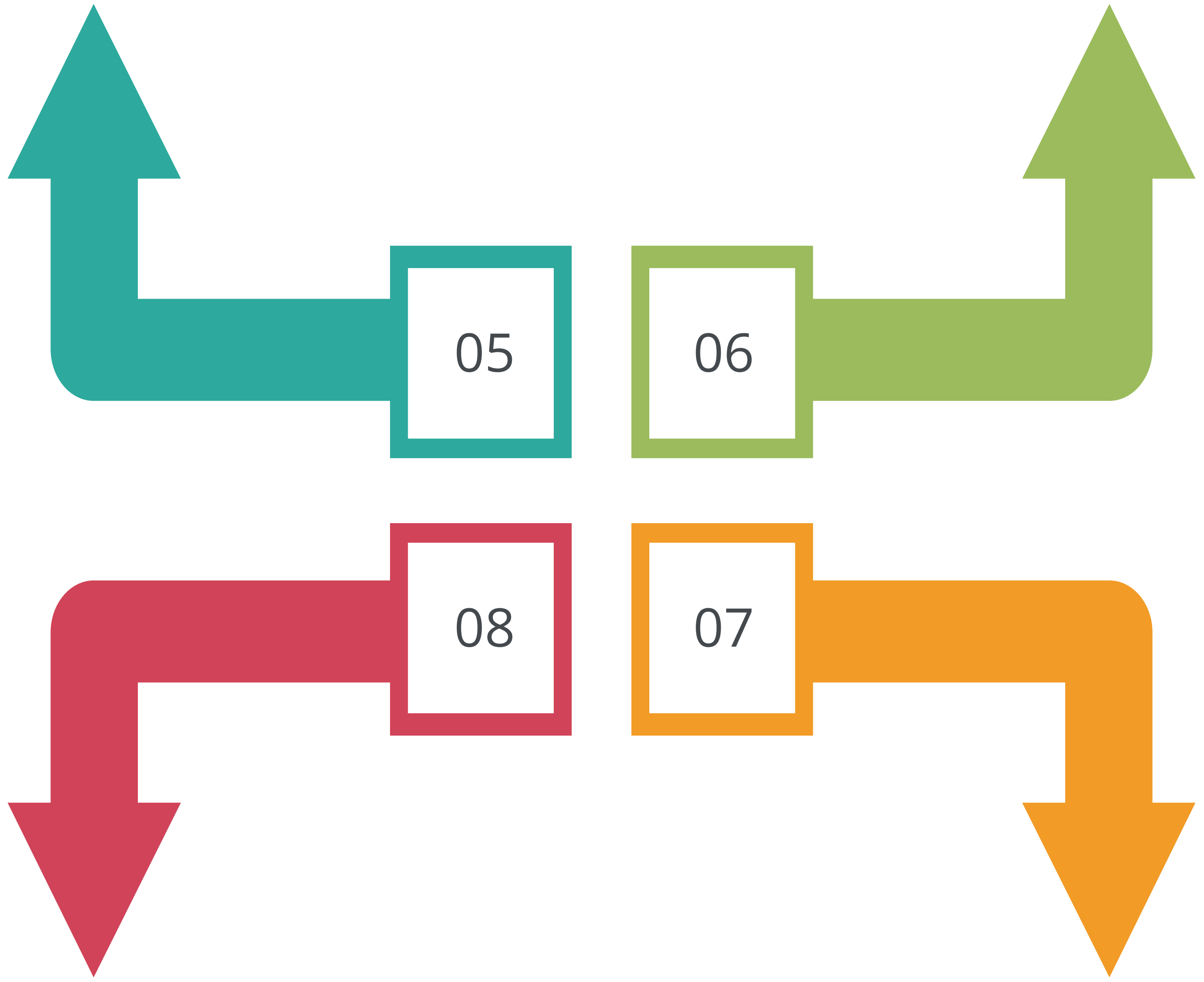
Playbook
Entry point of Ansible provisioning written in YAML

Module
Abstract of a system task like creating and changing files

Components of Ansible

Plays
Group of tasks that are carried out on specific hosts in order to implement specified functions (Playbook contains plays)

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

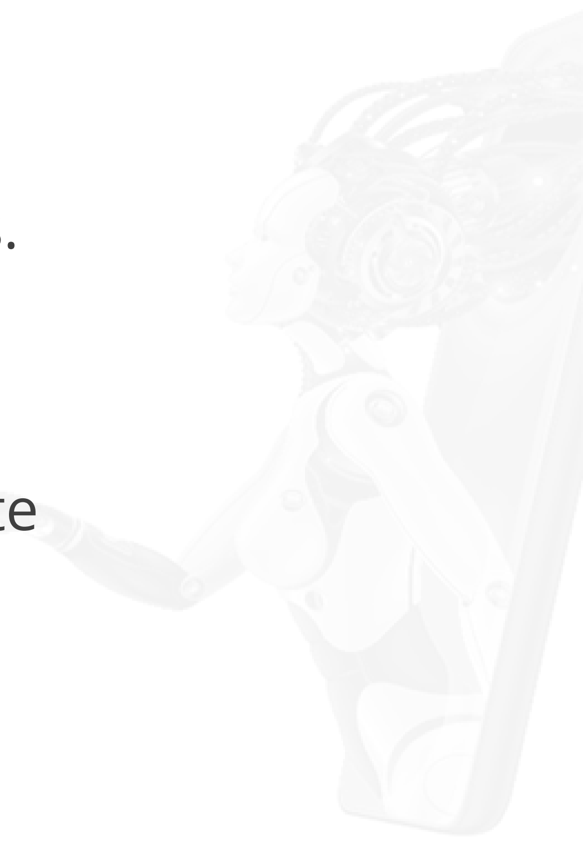


Facts
Global variables containing informations about the system

Handlers
Tasks that trigger changes in service status

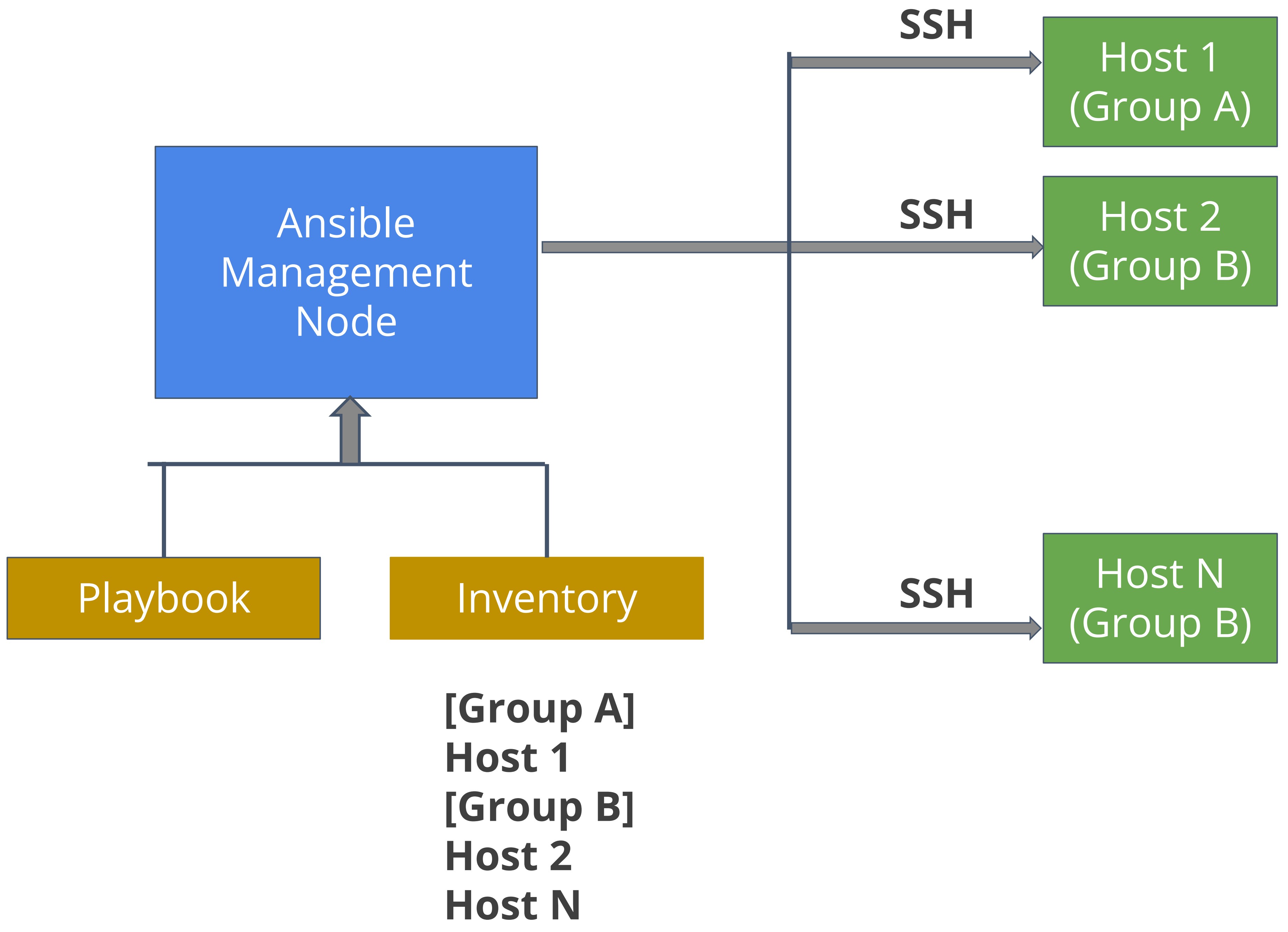
Working of Ansible

- 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.
- 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.



Working of Ansible

Below is the diagram representing the working of Ansible:



Introduction to Playbooks

Ansible's playbooks are one of the most important features, since they tell it what to execute.

- Playbooks are written in YAML and are easy to read, write, share, and understand.

- Each playbook contains one or more plays in a list.

Ansible can help in:

- Configuring declaration
- Orchestrating the steps of any manual procedure on multiple machines in a predetermined order
- Synchronous or asynchronous task execution

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 Playbook

Hosts and users:

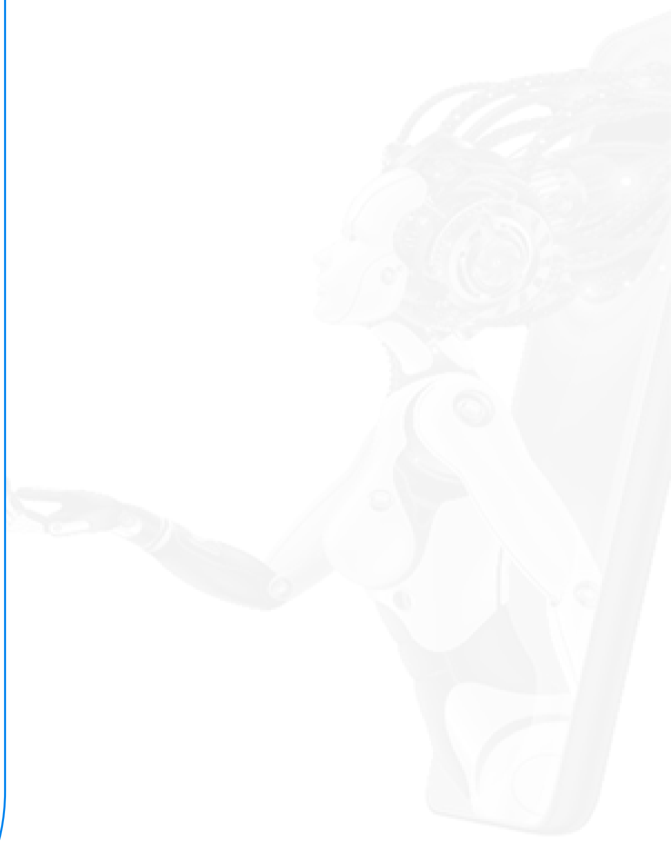
- For each play in a playbook, a target infrastructure is selected on which the tasks are executed.
- The host line is a list of one or more groups or host patterns separated by colons.
- The **remote_user** command refers to the name of the user account.

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

Components of Playbook

Plays and tasks:

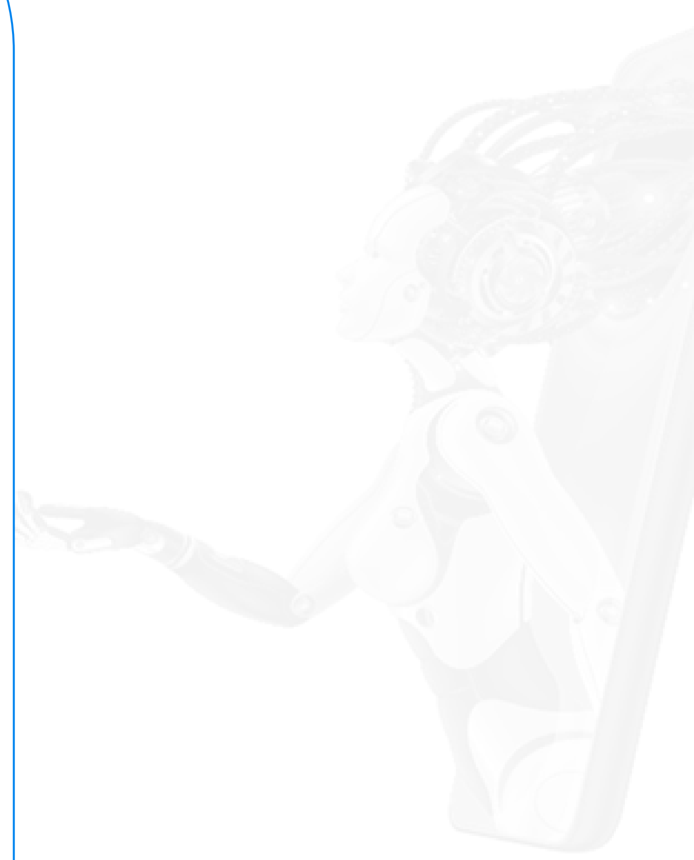
- Each play contains a list of tasks.
- Task is nothing but small operations that are executed on the target machine.
- Tasks are executed one at a time in an order.
- Within a play, all hosts get the same task directives.
- A play maps the hosts to the corresponding tasks.



Components of Playbook

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 Playbook

Ansible handlers

- Handlers are the **notify** actions that are triggered at the end of each block of tasks in a play.
- It only triggers 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 Playbook

Variables and roles

- Variables allow dynamic play content and reusability through multiple inventories.
- Roles are a way of loading certain vars_files, tasks, and handlers automatically based on a known file structure, allowing for easy sharing of roles with other users.

Introduction to Inventory

The inventory is a list or group of lists that Ansible uses to run against multiple controlled hosts in the infrastructure at the same time.

- The default location for inventory is a file called **/etc/ansible/hosts**.
- A different inventory file at the command line can be specified using the **-i <path>** option.
- 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.

Inventory Basics

Below is the sample 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

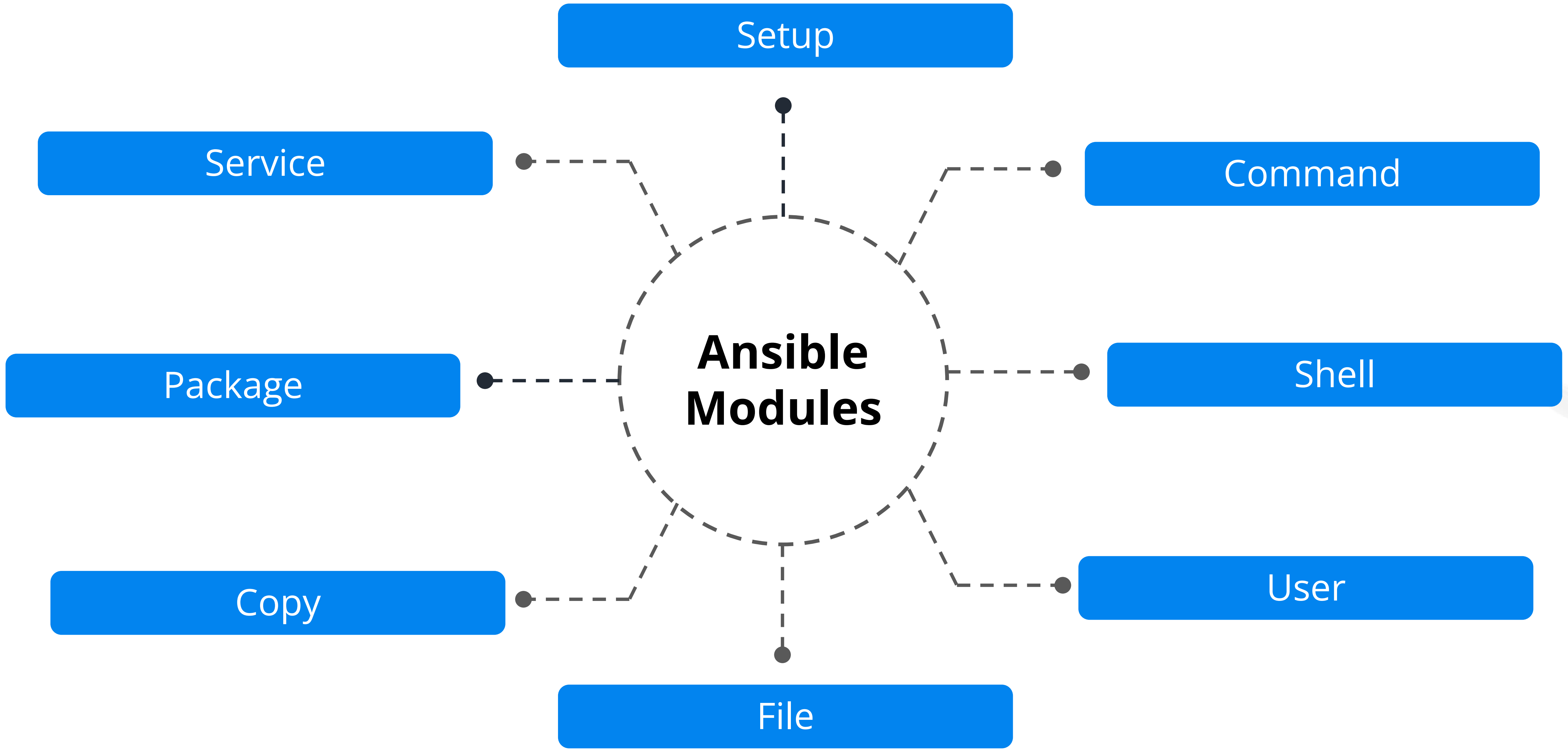
Groups of groups and group variables

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
```

Types of Modules

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



Working with 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
```

2) 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.

Ansible Variables

In Ansible, variables are used to handle device variations. Configuration file templates that are mostly the same, varies depending on the variables.

Variable Naming Convention:

- Letters, numbers, and underscores can be used as variable names. A letter should always be the first character in a variable.

  exp_12 is a best example of variable name. exp12 is good.

  exmp, exm p, 12, and exm.p are not valid variable names.

Ansible Roles

Based on a known file structure, roles allow you to load related vars files, tasks, handlers, and other Ansible objects automatically.

- Grouping the content by roles and 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

An Ansible Role has a standard directory structure with seven key directories. Each of these directories must be included in at least one role. Any directories not in use can be left out.

```
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:** Contain the list of processes to be executed
- **Handlers:** Contain 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
- **Files:** Contain the list of files to be deployed by the role
- **Templates:** Contain the templates to be executed by the role
- **Meta:** Contains general information about the role

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.

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.
- One can determine the tasks which can be skipped or run on particular nodes, with the help of conditional statements.

Demonstrating YAML Scripting

Duration: 20 Min.

Problem Statement: You are given a project to demonstrate YAML scripting.

Prerequisites:

- Ansible should be installed in your system.

Assisted Practice: Guidelines

Steps to demonstrate YAML scripting:

1. Create a playbook
2. Add YAML script to the playbook to install node
3. Run Ansible YAML script

Setting up Apache Web Server Using Ansible

Duration: 20 Min.

Problem Statement:

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

Assisted Practice: Guidelines

Steps to set up Apache server with Ansible:

1. Install Ansible on Ubuntu
2. Establish connectivity between Ansible controller and node machine
3. Create an Ansible playbook to install Apache web server
4. Run the Ansible playbook

Using Ansible Modules

Duration: 15 Min.

Problem Statement: You are given a project to create an Ansible program to implement modules.

Prerequisites: Ansible should e available in your system.

Assisted Practice: Guidelines

Steps to perform:

1. Execute an Ansible module on a local server

Creating and Working with Ansible Roles

Duration: 25 Min.

Problem Statement: You are given a project to create and work with Ansible Roles.

Prerequisites: Ansible should be installed in your system.

UNASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to create and work with Ansible Roles:

1. Install an Ansible and set up connectivity with the node machine
2. Create Ansible Role
3. Create Ansible tasks
4. Create Ansible Template
5. Create Ansible Variable
6. Remove unwanted directory
7. Create Ansible role playbook
8. Deploy Ansible role playbook

TECHNOLOGY

Terraform

simplilearn

Terraform

Terraform is a tool for building and versioning infrastructure efficiently. Terraform can manage existing service provider solutions.

- Terraform generates a plan describing what it will do to build the described infrastructure.
- Terraform determines the changes and creates an incremental execution plan, when configuration changes.
- Terraform can manage low-level components such as compute instances and storage along with high-level components such as DNS entries and SaaS features.

Key Features of Terraform

Infrastructure as a code
This allows a blueprint of the datacenter to be versioned.

Change automation
Change sheets in Terraform provide information and sequence of what changes will be made.

Execution plans
It shows what Terraform will do when you call apply.

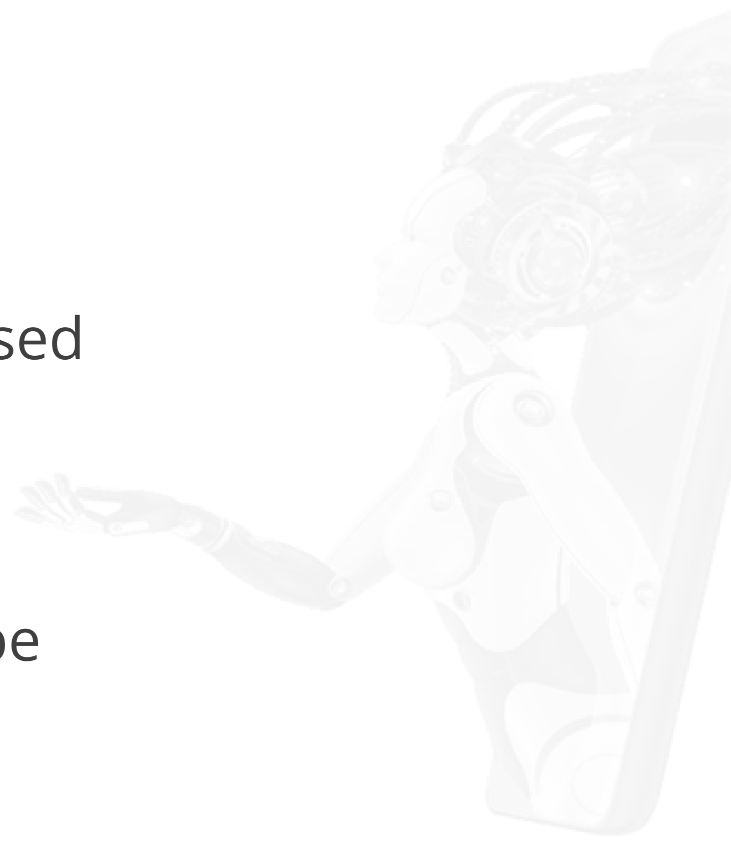
Resource graph
It is a graph of all resources which paralyzes any non-dependent resource.

Terraform Use Cases

- **Heroku app setup:** It is PaaS for hosting applications. It is scalable with the help of multiple dynos or workers.
- **Multi-tier applications:** Terraform handles multi-tier applications as a group of resources and all the dependencies are handled automatically.
- **Self-service cluster:** Scaling and building of a service can be converted into code and the service is handled automatically.
- **Software demos:** You can provide a Terraform configuration to create, provision, and bootstrap a demo on cloud providers like AWS.

Terraform Use Cases

- **SDN:** Terraform can be used to codify the configuration for SDNs.
- **Resource schedulers:** Resource schedulers allow Terraform to be used to set up infrastructure and provision the schedulers.
- **Multi-cloud deployment:** Terraform allows single configuration to be used to manage multiple providers and cloud dependencies.



Terraform Workflow

Terraform's core workflow consists of three steps:

- **Write:** Author infrastructure as code
- **Plan:** Preview changes before applying
- **Apply:** Provision reproducible infrastructure

Write

Terraform configuration is written in the same way as code.

```
---
# Create repository
$ git init my-infra && cd my-infra

Initialized empty Git repository in /.../my-
infra/.git/

# Write initial config
$ vim main.tf

# Initialize Terraform
$ terraform init

Initializing provider plugins...
# ...
 Terraform has been successfully

 initialized!

 # Make edits to config

 $ vim main.tf

 # Review plan

 $ terraform plan

 # Make additional edits, and repeat

 $ vim main.tf
```

Terraform Workflow

Plan

The final plan comes into play once the Write step's feedback loop has yielded a shift that looks fine.

```
$ git add main.tf

$ git commit -m 'Managing infrastructure
as code!'


[master (root-commit) f735520] Managing
infrastructure as code!

 1 file changed, 1 insertion(+)

$ terraform apply

An execution plan has been generated and
is shown below.

# ...
```

Terraform Workflow

Apply

Terraform provisions the real infrastructure after one last step.

```
Do you want to perform these actions?

  Terraform will perform the actions
described above.

  Only 'yes' will be accepted to approve.

  Enter a value: yes

# ...

Apply complete! Resources: 1 added, 0
changed, 0 destroyed.

$ git remote add origin
https://github.com/*user*/*repo*.git

$ git push origin master
```

File Extension

Simple text files with the **.tf** extension are used to store Terraform code. A JSON-based version of the language with the .tf.json file extension is also available.

Text Encoding

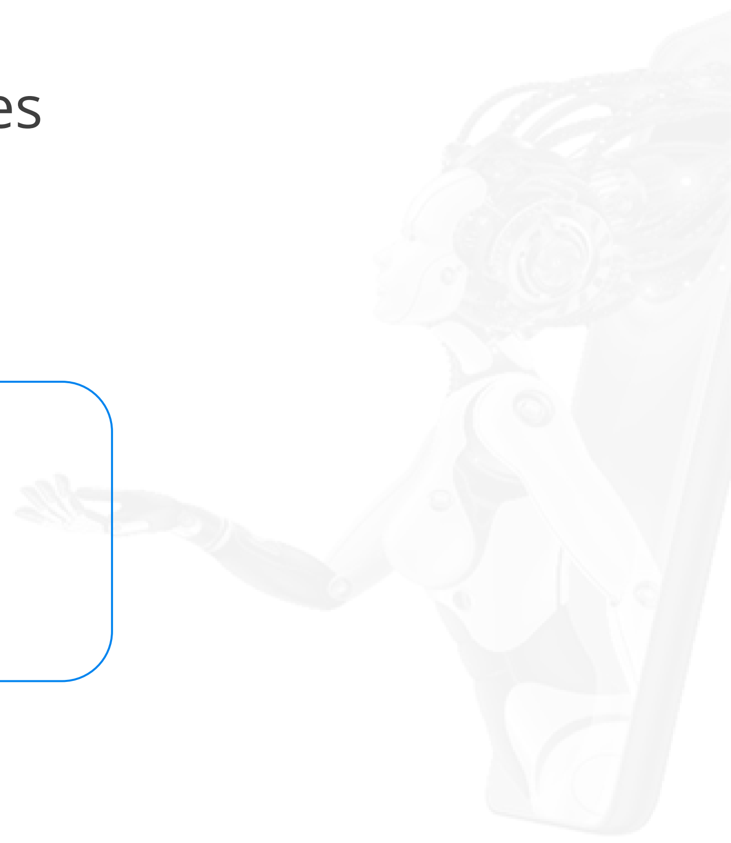
UTF-8 encoding must be used in all configuration files.

Directories and Modules

- A module is a directory containing a list of.tf and/or.tf.json files.
- Terraform tests all module's configuration files, essentially treating the whole thing as a single text.
- Module calls allow Terraform modules to directly include other modules in their configuration.

Root Module

The working directory where Terraform is invoked is the called the root module.



Terraform Commands

The `terraform` command provides a command-line interface to Terraform.

Note

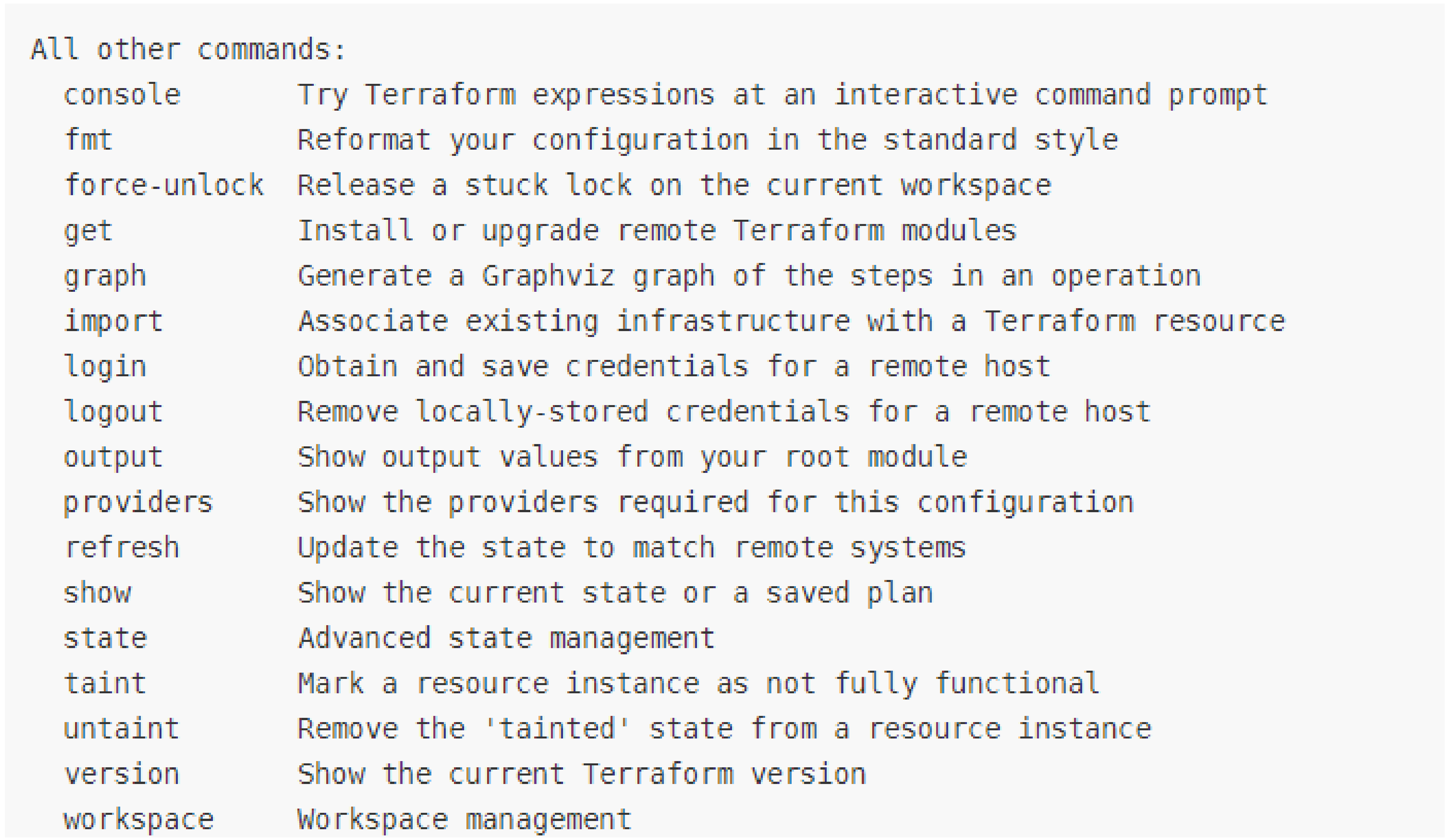
- Run terraform with no arguments to see a list of the commands available in your current Terraform version:
- Use the -help option with the appropriate subcommand to get precise help with any command.

ex: to see help about the *validate* subcommand you can run `terraform validate -help`

Main Commands

Command	Use
`init`	To set up your working directory to run any additional commands
`validate`	To validate the configuration
`plan`	To view the changes required by the current configuration
`apply`	To set up or update infrastructure
`destroy`	To delete previous infrastructure

Other Commands



```
All other commands:
  console       Try Terraform expressions at an interactive command prompt
  fmt           Reformat your configuration in the standard style
  force-unlock  Release a stuck lock on the current workspace
  get           Install or upgrade remote Terraform modules
  graph         Generate a Graphviz graph of the steps in an operation
  import        Associate existing infrastructure with a Terraform resource
  login         Obtain and save credentials for a remote host
  logout        Remove locally-stored credentials for a remote host
  output        Show output values from your root module
  providers     Show the providers required for this configuration
  refresh       Update the state to match remote systems
  show          Show the current state or a saved plan
  state         Advanced state management
  taint         Mark a resource instance as not fully functional
  untaint       Remove the 'tainted' state from a resource instance
  version       Show the current Terraform version
  workspace     Workspace management
```

source: https://www.terraform.io/

Setting Up Terraform

Duration: 15 Min.

Problem Statement:

You are given a project to set up Terraform in your local system.

Assisted Practice: Guidelines

Steps to set up Terraform in your local system:

1. Install Terraform on Ubuntu

Key Takeaways

- Configuration management tools manage all configuration items in a software for all environments.
- Ansible's core functionality is augmented by plugins.
- An Ansible playbook is a blueprint for automation tasks, wherein Ansible inventory is a file that contains information about the managed hosts.
- Ad-hoc commands are ideal for tasks that are only performed rarely

Key Takeaways

- Ansible uses YAML to express Ansible playbooks.
- Role is the basic process for breaking a playbook into multiple files.
- Ansible facts are system properties collected by Ansible while executing tasks on a machine.
- Terraform generates a plan describing what it will do to build the described infrastructure

Lesson-End Project

Provision EC2 using Terraform

Project Agenda: To automate provisioning of an AWS EC2 instance using Terraform.

Description: You are a DevOps Engineer in an IT company. Your company wants you to automate an infrastructure to manage all the running services from one place and provide an access model control based upon the organisation, teams, and users.This should also help the team in better collaboration and a centralised documentation**.**

Perform the following:
Configure AWS in your local system and launch an EC2 instance using Terraform.