Caltech | Center for Technology & Management Education

Post Graduate Program in DevOps

Powerd by simplilearn

DevOps and Cloud Computing

Caltech | Center for Technology & Management Education

Configuration Management with Ansible and Terraform

Powerd by simplilearn

Writing Ansible Playbooks

A Day in the Life of a DevOps Engineer

You are working in an organization that uses Ansible for configuration management.

You are part of a DevOps team that is looking for an Ansible feature where an ordered list of tasks can be saved in a file that can be used to repeatedly run those tasks.

The team also wants the Ansible setup, deployment, and orchestration functions to be recorded. Additionally, they want to manage the configurations and deployments to remote machines.

The organization also wants to help employees deal with the differences between systems, as no two systems are precisely alike.

A Day in the Life of a DevOps Engineer

Additionally, the company is also looking for a solution to generate accurate reports or expected results.

The team would also require an option that will allow a task to only run when called. For example, when an update is made on the managed host, it should act.

To achieve all the above, along with some additional features, you will be learning a few concepts in this lesson that will help find a solution for the given scenario.

Learning Objectives

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

- Configure Ansible playbook
- Analyze task iterations with loops
- Understand conditionals in Ansible playbook
- Configure Ansible handler

Introduction to Playbook

What Is a Playbook?

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

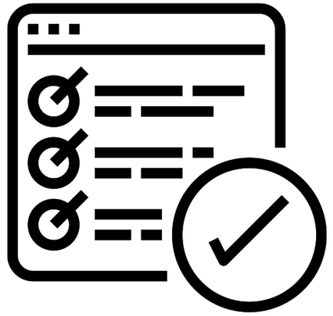
ANSIBLE

Playbooks

Playbook Features

These are the features of Ansible:

- 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 for mapping the hosts to well-defined roles represented by Ansible tasks.
- A **play** can also be used to orchestrate multiple machine deployment and running processes on target machines.



What Can a Playbook Do?

- Ansible setup, deployment, and orchestration functions are recorded and executed via playbooks.
- Playbooks can be used to manage configurations and deployments to remote machines.
- They can also sequence multi-tier rollouts involving rolling updates and can delegate actions to other hosts.

How to Write an Ansible Playbook?

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
```

How to Write an Ansible Playbook?

Every playbook breaks down into the same standard sections:

PLAYBOOK

How to Write an Ansible Playbook?

Playbooks begin with three hyphens "**---**".

```
---
-  name: Example Playbook
   hosts: all
   become: yes

   vars:
       greeting: Hello World!

   tasks:
   - name: Creating a New Directory
        file:
          path: "/home/example_playbook"
          state: directory
   - name: Deploy Greeting
        copy:
      dest: "/home/example_playbok"
          content: "{{ greeting }}"
...
```

How to Write an Ansible Playbook?

Host: The host section defines the target machines where the playbook will run.

```
---
-  name: Example Playbook
   hosts: all
   become: yes

   vars:
       greeting: Hello World!

   tasks:
   - name: Creating a New Directory
        file:
          path: "/home/example_playbook"
          state: directory
   - name: Deploy Greeting
        copy:
      dest: "/home/example_playbok"
          content: "{{ greeting }}"
...
```

How to Write an Ansible Playbook?

Variable: The variable section is optional and includes any variables that the playbook requires.

```
---
-  name: Example Playbook
   hosts: all
   become: yes

   vars:
       greeting: Hello World!

   tasks:
   - name: Creating a New Directory
        file:
          path: "/home/example_playbook"
          state: directory
   - name: Deploy Greeting
        copy:
      dest: "/home/example_playbok"
          content: "{{ greeting }}"
...
```

How to Write an Ansible Playbook?

Tasks: The task section lists all tasks that the target machine must run and specifies the use of Modules.

```
---
-  name: Example Playbook
   hosts: all
   become: yes

   vars:
       greeting: Hello World!

   tasks:
   - name: Creating a New Directory
        file:
          path: "/home/example_playbook"
          state: directory
   - name: Deploy Greeting
        copy:
      dest: "/home/example_playbok"
          content: "{{ greeting }}"
...
```

How to Write an Ansible Playbook?

Most playbooks end with three periods **"..."**.

```
---
-  name: Example Playbook
   hosts: all
   become: yes

   vars:
       greeting: Hello World!

   tasks:
   - name: Creating a New Directory
        file:
          path: "/home/example_playbook"
          state: directory
   - name: Deploy Greeting
        copy:
      dest: "/home/example_playbok"
          content: "{{ greeting }}"
...
```

Points to Remember

Playbooks are written in the YAML format and have a **.yml** file extension.

Command to run a playbook:
$ ansible-playbook <playbook.yml>

Create lists:
"–" symbol is used to create a list.

Command to check syntax errors:
$ ansible-playbook <playbook.yml> --syntax-check

Points to Remember

Include whitespaces

Name your tasks

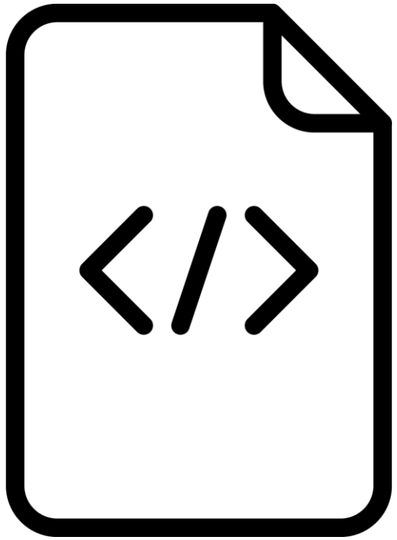
Include the state

Employ comments

Playbook Syntax

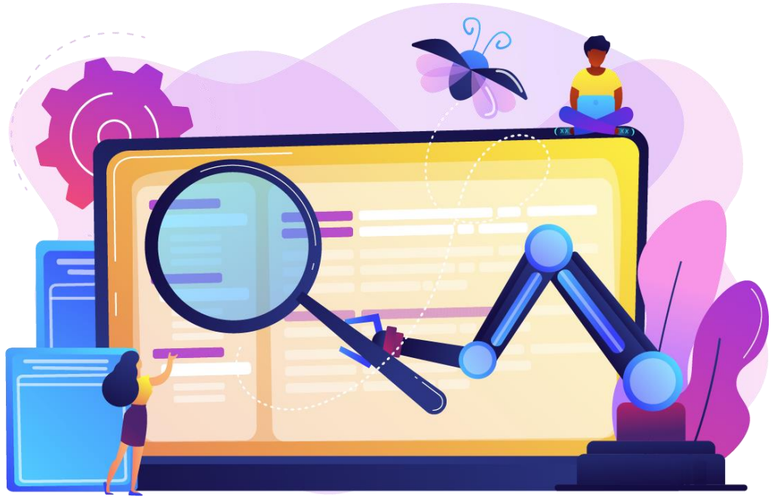
Playbook consists of one or more **plays** in a logical order.

- Each play carries out a portion of the playbook's main objective by carrying out one or more responsibilities.
- Each task seeks an Ansible module.



Playbook Execution

A playbook is executed in a sequence. Within a play, tasks are also executed from top to bottom.



In case of multiple plays, the playbook can manage multi-machine deployments with one play operating on servers, the other one on database servers, one on network infrastructure, and many more.

Components of Playbooks

Plays and Tasks:

01 Each play contains a list of tasks.

02 Tasks are nothing but small operations that are executed on the target machine.

03 Tasks are executed one at a time in order.

04 Within a play, all hosts get the same task directives.

05 A play maps the hosts to the corresponding tasks.

Components of Playbooks

Plays and Tasks:

06 During playbook execution, hosts with failed tasks are taken out of the rotation for the entire playbook.

07 The goal of a task is to execute a module with specific arguments.

08 To achieve a short execution time, modules should be idempotent.

09 It is recommended to check the module's state. If it's true, then the final state has been achieved and the execution can be stopped.

10 Rerunning the plays becomes idempotent if the modules are idempotent.

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 according to the task as shown below:

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

Components of Playbooks

- Use the keyword **become** on a particular task to change the user account.

```
---

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

Assisted Practice

Creating Your First Playbook

Duration: 15 Min.

Problem Statement:

Nodejs is an open-source, cross-platform that executes JavaScript code outside of a web browser. You have been given a task to write a playbook consisting of instructions to install Nodejs.

Assisted Practice: Guidelines

Steps to be followed:

1. Create a Playbook
2. Run Ansible YAML script

Running the Playbook

Options to Trigger While Executing a File

1. Ask for the vault password:
   --ask-vault-password, --ask-vault-pass

2. Run operations as this user:
   --become-user <BECOME_USER>

3. Clear the fact cache for every host in inventory:
   --flush-cache

4. Run handlers even if a task fails:
   --force-handlers

5. Output a list of matching hosts without executing anything else:
   --list-hosts

Source: www.docs.ansible.com

Options to Trigger While Executing a File

6 List all available tags:
--list-tags

7 List all the tasks to be executed:
--list-tasks

8 Run plays and tasks whose tags do not match these values:
--skip-tags

9 Confirm each task before running:
--step

10 Perform a syntax check on the playbook without executing it:
--syntax-check

Options to Trigger While Executing a File

11 The vault identity to use:
--vault-id

12 Vault password file:
--vault-password-file, --vault-pass-file

13 Ask for privilege escalation password:
-K, --ask-become-pass

14 Run operations with become :
-b, --become

15 Override the connection timeout in seconds:
-T <TIMEOUT>, --timeout <TIMEOUT>

Using Variables in Playbook

What Are Variables?

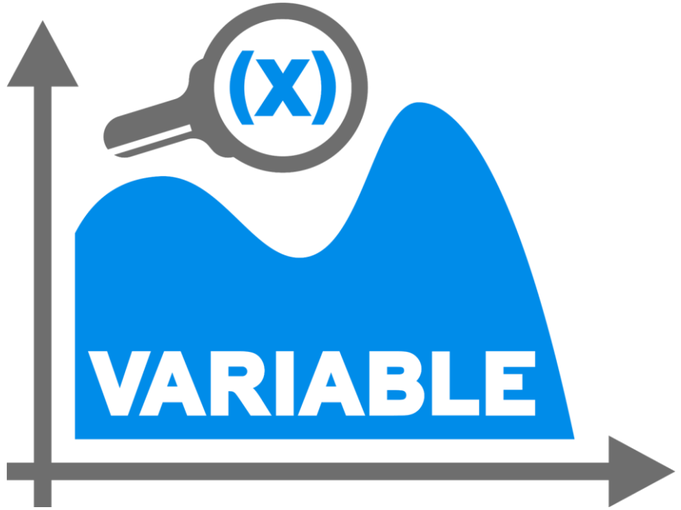
Ansible uses variables to help users deal with the difference between systems, as no two systems are precisely alike.

Variables:

```
foo
foo_env
foo_port
foo5
```

Creating a Variable Name

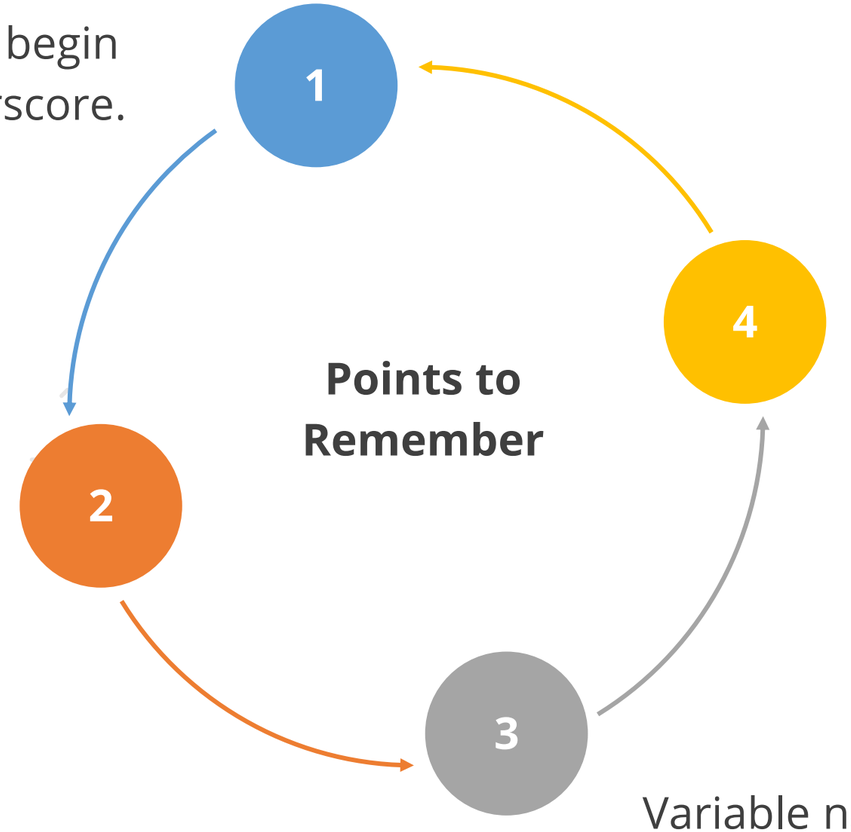
A variable name comprises letters, numbers, and underscores, or a combination of all.



Variables are defined directly in playbooks by using the “vars:” command.

Creating a Variable Name

Following are some points to remember while creating a variable name :



1. A variable name should begin with a letter or an underscore.
2. A variable name cannot have a number as the first character.
3. Variable names cannot have blank spaces.
4. Variable names containing playbook or python keywords are not valid.

Creating a Variable Name

- Examples of valid variable names:

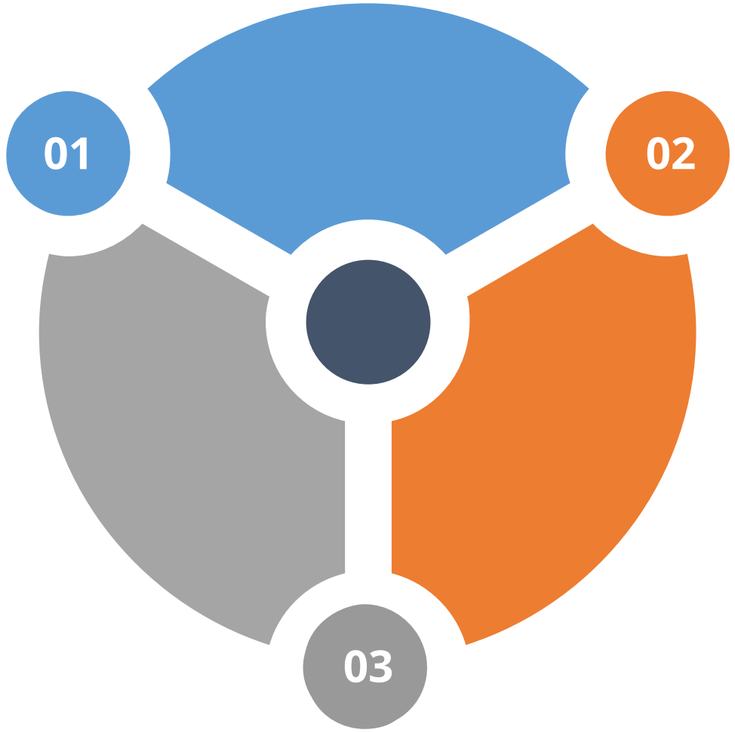
```
basketball
basket_ball
basketball321
basket_ball321
```

- Examples of invalid variable names:

```
basket ball
123
123_basketball
 basket-ball
```

Types of Variables

Following are some of the variable types:



Simple Variables

List Variables

Dictionary Variables

Simple Variables

A variable name with a single value is known as a Simple variable.

They can be defined using standard YAML syntax.

```
remote_install_path: /opt/my_app_config
```

List Variables

A variable name with multiple values is known as a List variable. Multiple values can be saved as an itemized list or inside square brackets [ ], divided by commas.

They can be defined using standard YAML syntax.

```
region:
  - northeast
  - southeast
  - midwest
```

Dictionary Variables

In dictionaries, data such as ID information or a user profile is stored.

- Complex variables can be defined using YAML dictionaries.
- Keys can also be mapped to values by using a YAML dictionary.

```
foo:
  field1: one
  field2: two
```

Defining Variables in a Play

Variables can be defined directly in a play.

```
- hosts: webservers
  vars:
    http_port: 80
```

Variables defined in a play are only visible to tasks executed in that play.

Defining Variables in Inventory

The inventory file contains the list of the managed nodes. Sometimes, it is also called the **host file.**

The inventory file also organizes managed nodes, creating and nesting groups for scaling.

Host and Groups:

- The format of the inventory file depends on the plugin present in the system.
- The most common formats are INI and YAML.

Defining Variables in Inventory

Below is the difference between INI and YAML inventory files:

INI format:

```
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
```

YAML format:

```
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
```

Defining Variables in Inventory

Host Variables:

Below mentioned are some examples of assigning the variables to the hosts:

```
[atlanta]

host1 http_port=80 maxRequestsPerChild=808

host2 http_port=303 maxRequestsPerChild=909
```

Defining Variables in Inventory

Group Variables:

Applies variables to an entire group at once, as shown below:

INI version:

```
[atlanta]
host1
host2

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

YAML version:

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

Defining Variables in Inventory

Groups of Groups

To make groups of groups, use the **:children** suffix in INI or the **children:** entry in YAML.

Group Variables

To apply variables to groups of groups, use **:vars or vars:**

Defining Variables in Inventory

INI Version:

```
[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 version:

```
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:
```

Defining Variables in Inventory

Properties of child groups:

- A child member is automatically a member of the parent group.
- A child's variable will have higher precedence than a parent's variable.
- Groups can have multiple parents and children, but it cannot be bidirectional.
- If a host is present in multiple groups, only one instance of a host can gather data from other instances of the same host.

Assisted Practice

Executing Variables in a Playbook

Duration: 15 Min.

Problem Statement:

You have been assigned a task to define variables in the standard YAML syntax, to represent the variations. Further, you need to execute the playbook containing variables.

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a Playbook
2. Executing the Playbook

Task Iterations with Loops

Ansible Loops

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

Ansible offers three keywords for creating loops:

1. loop
2. with_<lookup>
3. until

An Ansible loops include changing ownership for several files or directories, or both, with the file module, creating multiple users, and repeating a step.

Working with Loops

- The **with_<lookup>** is dependent on lookup plugins.
- The loop keyword is similar to **with_list**.
- The loop keyword does not accept a string input.
- Use **with_items** to perform 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:

1. **Iterating over a simple list:**

- Define the list directly in the task:

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

Working with Loops

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

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

Working with Loops

2. **Iterating over a list of hashes:**

- With a list of hashes, refer to the subkeys 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

3. **Iterating over a dictionary:**

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

```
- name: Using dict2items
  ansible.builtin.debug:
    msg: "{{ item.key }} - {{ item.value }}"
  loop: "{{ tag_data | dict2items }}"
  vars:
    tag_data:
      Environment: dev
       Application: payment
```

Working with Loops

Complex loops:

1. **Iterating over nested lists:**

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

```
- name: Give users access to multiple databases
  community.mysql.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

2. **Retrying a task until a condition is met:**

- Use the keyword **until** to retry a process till the condition is met.

  For example:

```
- name: Retry a task until a certain condition is met
  ansible.builtin.shell: /usr/bin/foo
  register: result
  until: result.stdout.find("all systems go") != -1
  retries: 5
  delay: 10
```

Working with Loops

3. Looping over inventory

- With **ansible_play_batch** or **groups** variables, use a regular loop.

  For example:

```
- name: Show all the hosts in the inventory
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ groups['all'] }}"

- name: Show all the hosts in the current play
  ansible.builtin.debug:
    msg: "{{ item }}"
   loop: "{{ ansible_play_batch }}"
```

Working with Loops

- There is a distinct lookup plugin **inventory_hostnames**.

  For example:

```
- name: Show all the hosts in the inventory
  ansible.builtin.debug:
    msg: "{{ item }}"
  loop: "{{ query('inventory_hostnames', 'all') }}"

- name: Show all the hosts matching the pattern, ie all but the group www
  ansible.builtin.debug:
    msg: "{{ item }}"
   loop: "{{ query('inventory_hostnames', 'all:!www') }}"
```

Working with Loops

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

01 **register:** To register a variable using loops

02 **until:** To repeat a task until a condition is met

03 **ansible_play_batch:** To use a loop with inventory

04 **label** and **loop_control:** To limit the output of the loop

05 **pause:** To stop the execution for a short period between different tasks

Working with Loops

Below is the list of variables used using the **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)

Working with Loops

Variable	Description
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

Assisted Practice

Executing Loops in a Playbook

Duration: 15 Min.

Problem Statement:

You have been assigned a task to define and execute loops in the playbook.

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a Playbook
2. Executing the Playbook

Conditional Tasks with Ansible

Conditionals in Ansible Playbook

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

Conditionals in Ansible Playbook

The following are some uses of conditional statements:

01 Conditional statements are used in playbooks where there are multiple variables representing different entities such as software packages and servers.

02 Conditional statements helps in controlling the flow of execution.

03 Conditional statements are used to identify tasks that can be skipped or that should be executed on specific nodes.

Conditionals in Ansible Playbook

Most common ways of implementing Ansible conditionals are:

01 The when statement

02 Loops and conditionals

03 Custom facts

04 Register variables

05 Conditional imports

The When Statement

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

```
tasks:
  - name: "shut down Debian flavored systems"
    command: /sbin/shutdown -t now
    when: ansible_facts['os_family'] == "Debian"
```

Note

All variables can be used directly in conditionals without using double curly braces.

The When Statement

Parentheses can be used 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")
```

The When Statement

- When statement can also make use of a variety of Jinja2 "tests" and "filters."
- Ignore one statement's error and then decide whether or not to do something based on success or failure.

Example

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

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

  - command: /bin/something_else
    when: result is succeeded

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

Loops and Conditionals

- Process the conditional statement independently for each item in the loop
- Example of **when** statement with loop:

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

- Use the default filter to give a blank iterator to ignore the entire task based on the loop variable, as illustrated:

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

Loops and Conditionals

Here is an example of using a dict in a loop:

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

Custom facts

Here is an example of **when** statement with custom facts:

```
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'
```

Custom facts

When statement applied to includes, imports, and roles:

- It is recommended to group multiple tasks if they share the same conditional statement.
- In these instances, all tasks are evaluated, but the conditional is applied to each task as shown below:

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

Register Variables

- It's common in a playbook to save the outcome of a command in a variable and then retrieve it later.

- The register keyword determines where a result should be saved. Action lines, templates, and **when** statements can all benefit from the resulting variables.

```
- name: test play
  hosts: all

  tasks:

      - shell: cat /etc/motd
        register: motd_contents

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

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

Using conditionals in Ansible Playbook

Duration: 15 Min.

Problem Statement:

You have been assigned a task to define and execute conditionals in the playbook with which you can execute different tasks, or have different goals, depending on the value of a fact (data about the remote system), a variable, or the result of a previous task.

Assisted Practice: Guidelines

Steps to be followed:

1. Running playbook containing conditionals

Ansible Handlers

Ansible Handlers

A handler is similar to any other task, but it only runs when it is called or informed. When an update is made on the managed host, it takes action.

Ansible Handlers

- Handlers are the **notify** actions that are triggered at the end of each block of tasks in a play.
- They are 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
```

Ansible Handlers

Following are some uses of Handlers:

- After installation, handlers are used to initiate a secondary update, such as starting or restarting a service.

- They are also useful in reloading a service after some changes have been made to the configuration files.

Handler Example

An example of a playbook **verify-apache.yml** that contains a play with a handler:

```
---
 - name: Verify apache installation
   hosts: webservers
   vars:
     http_port: 80
     max_clients: 200
   remote_user: root
   tasks:
   - name: Ensure apache is at the latest version
     ansible.builtin.yum:
       name: httpd
       state: latest

  - name: Write the apache config file
     ansible.builtin.template:
       src: /srv/httpd.j2
       dest: /etc/httpd.conf
     notify:
    - Restart apache
```

Handler Example

An example of a playbook **verify-apache.yml** that contains a play with a handler:

```
- name: Ensure apache is running
   ansible.builtin.service:
     name: httpd
     state: started

 handlers:
   - name: Restart apache
     ansible.builtin.service:
       name: httpd
       state: restarted
```

Handler Example

The second task in this example playbook notifies the handler. More than one handler can be notified by a single task:

```
- name: Template configuration file
  ansible.builtin.template:
    src: template.j2
    dest: /etc/foo.conf
  notify:
    - Restart memcached
    - Restart apache
  handlers:
    - name: Restart memcached
      ansible.builtin.service:
        name: memcached
        state: restarted
    - name: Restart apache
      ansible.builtin.service:
        name: apache
        state: restarted
```

Controlling When Handlers Run

Handlers execute when all of the tasks in a play have been performed. This technique is efficient because the handler only executes once, irrespective of the number of processes that inform it.

Controlling When Handlers Run

Add a task to flush handlers using the **meta** module.
If you need them to run before the procedure is finished, proceed as follows:

```
tasks:
  - name: Some tasks go here
    ansible.builtin.shell: ...

  - name: Flush handlers
    meta: flush_handlers

  - name: Some other tasks
    ansible.builtin.shell: ...
```

Note

The **meta: flush_handlers** task activates the handlers that have been notified at that time in the play.

Variables with Handlers

Variables must not be named after the handler. Because handler names are templated early.

A value for a handler name might not be available in Ansible like this.

```
handlers:
- name: Restart "{{ web_service_name }}"
```

Note

The entire play will fail if the variable used in the handler name is not available.
Changing that variable in the mid-play will not lead to a new handler being created.

Variables with Handlers

Users can insert the variables in their Handler's task parameters.

Include_vars can be used to load values as shown:

```
tasks:
  - name: Set host variables based on distribution
    include_vars: "{{ ansible_facts.distribution }}.yml"

handlers:
  - name: Restart web service
    ansible.builtin.service:
      name: "{{ web_service_name | default('httpd') }}"
       state: restarted
```

Variables with Handlers

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"
```

Assisted Practice

Configuring Tasks with Handlers

Duration: 15 Min.

Problem Statement:

You have been assigned a problem, to execute a task only when a change is made on a machine, using Handlers. Each handler should have a globally unique name.

Assisted Practice: Guidelines

Steps to be followed:

1. Creating a playbook with handlers
2. Executing the Playbook

Key Takeaways

- Playbooks are an ordered list of tasks saved in a file, which can be used to repeatedly run those tasks.
- Ansible uses variables to help users deal with the differences between systems, as no two systems are precisely alike.
- Ansible conditionals are the control statements used in a playbook that generates an accurate report or expected output.
- Ansible handler is similar to any other task, but it only runs when it is called or informed.

Website Deployment Using Ansible

Duration: 25 Min.

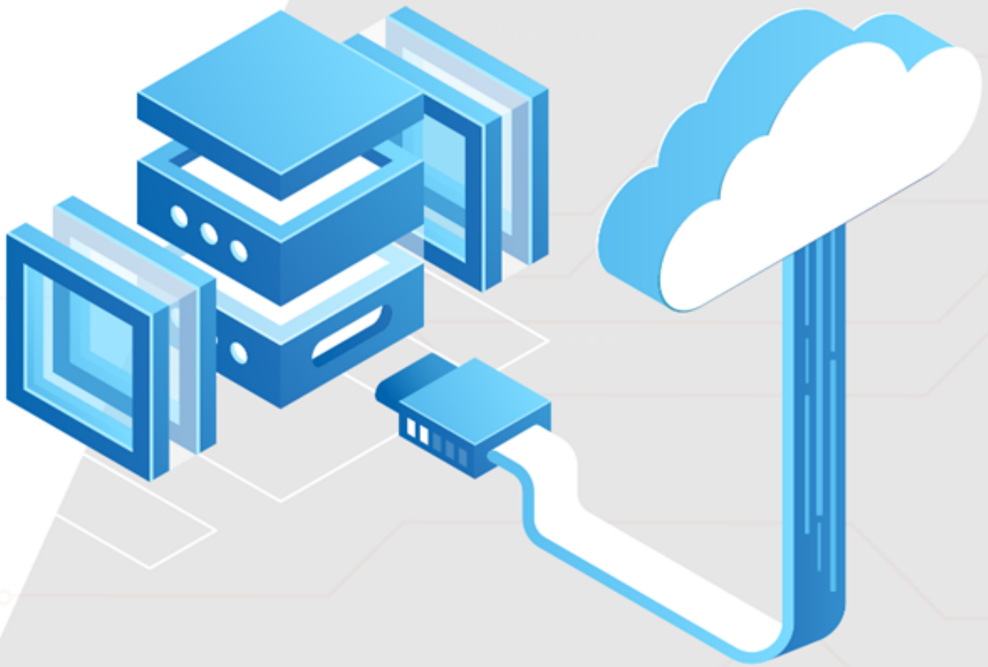
Project agenda: To deploy a static HTML website with Ansible

Description: Ansible Roles are a structured way of grouping tasks, handlers, vars, and other properties. They increase reusability. For this project, we will use two Ubuntu machines. The first one will be your Ansible controller and the second one will be your target machine for Apache installation.

Perform the following:

- Checking the connectivity of the target machine from the controller through Ansible
- Creating a Role under the Role folder
- Configuring the main components of Role
- Running the Playbook

LESSON-END PROJECT



Thank you