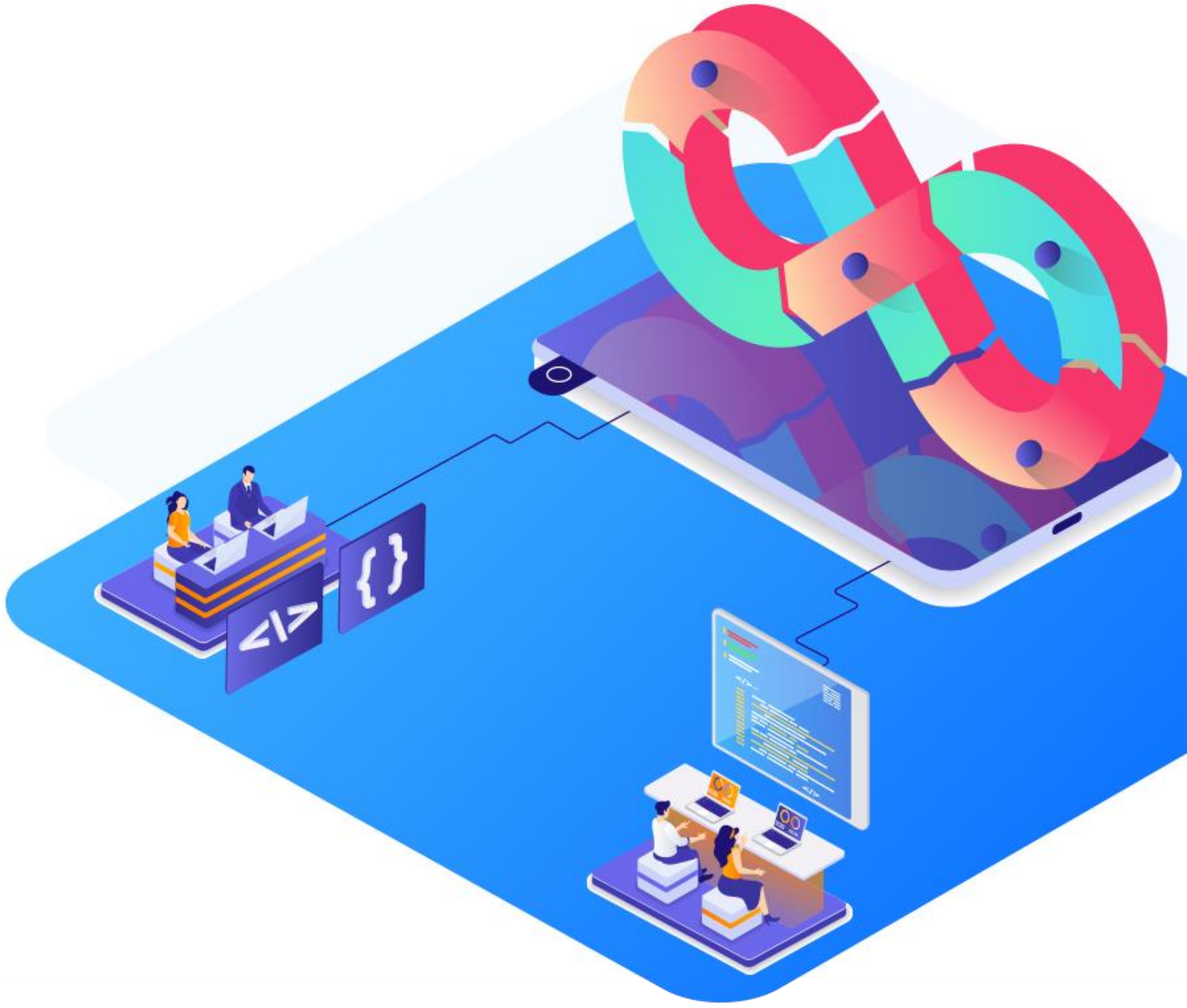
Configuration Management with Ansible and Terraform



Writing Ansible Playbooks

Learning Objectives

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

- Execute a basic Ansible playbook on a remote server to perform system configurations
- Implement an Ansible playbook to automate web server deployment
- Apply variables within an Ansible playbook to customize deployment configurations
- Use loops in an Ansible playbook to perform iterative tasks
- Apply conditional statements in an Ansible playbook to manage different deployment scenarios

Learning Objectives

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

- Assess the role of handlers in an Ansible playbook to understand their impact on system state management
- Outline the use of filters in an Ansible playbook to ensure accurate data processing and output customization

Introduction to Playbook

What Is a Playbook?

In Ansible, a playbook is a YAML (Yet Another Markup Language) script that defines tasks for remote hosts, enabling the automation of complex configurations in a human-readable format.

ANSIBLE

Playbooks

They consist of multiple plays, where each play maps a group of hosts to tasks that should be run on them, thus allowing the orchestration of multi-step processes across multiple systems.

Purpose of Playbook

Below are some use cases of the Ansible playbook:

Automation	Automate routine and complex tasks, eliminating manual intervention
Configuration management	Manage system configurations by specifying the desired state of system resources
Infrastructure as Code (IaC)	Define the desired state of the infrastructure and contain a sequence of tasks or actions to achieve that state

Playbook Syntax

Playbooks describe the tasks to be executed on managed hosts. The following YAML script illustrates the key components and syntax used in Ansible playbooks:

Example:

```
---
- name: Install and start Apache
  hosts: webservers
  become: yes

  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present

    - name: Ensure Apache is running
      service:
        name: apache2
        state: started
```

Components of Playbook

Name	Each playbook starts with a **name** field, providing a human-readable description of the playbook.
Hosts	The **hosts** field specifies the target hosts or groups of hosts on which the tasks in the playbook should be executed.
Become	The **become** field indicates whether to run tasks with elevated privileges, such as **sudo**.
Variables	The **vars** field defines variables that can be used throughout the playbook.

Components of Playbook

Tasks	The **tasks** field contains a list of tasks to be executed on the target hosts.
Handlers	The **handlers** field contains special tasks that can be notified by other tasks to perform actions such as restarting a service.
Roles	Roles organize playbooks into reusable components, including tasks, variables, handlers, and other elements.
Modules	Modules are the tools used by tasks to perform actions on managed hosts.

How to Write a Playbook?

A playbook begins with three dashes **(---)**

Example:

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

   vars:
    greeting: Hello World!

   tasks:
   - name: Creating a New Directory
     file:
       path: "/home/example_playbook"
       state: directory
```

How to Write a Playbook?

The **hosts** section defines the target machines where the playbook will run.

Example:

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

   vars:
    greeting: Hello World!

   tasks:
   - name: Creating a New Directory
     file:
       path: “/home/example_playbook”
       state: directory
```

How to Write a Playbook?

The **become: yes** value enables privilege escalation, allowing the task to be executed with sudo or root privileges.

Example:

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

   vars:
    greeting: Hello World!

   tasks:
   - name: Creating a New Directory
     file:
       path: "/home/example_playbook"
       state: directory
```

How to Write a Playbook?

The **vars** section is optional and includes any variables that the playbook requires.

Example:

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

    vars:
     greeting: Hello World!

    tasks:
    - name: Creating a New Directory
      file:
        path: “/home/example_playbook”
        state: directory
```

How to Write a Playbook?

The **tasks** section lists all tasks that the target machine must run and specifies the use of modules.

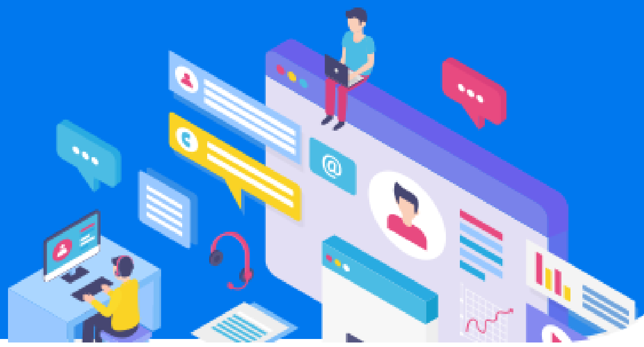
Example:

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

   vars:
    greeting: Hello World!

   tasks:
   - name: Creating a New Directory
     file:
       path: "/home/example_playbook"
       state: directory
```

Assisted Practice



Creating an Ansible playbook

Duration: 20 Min.

Problem statement:

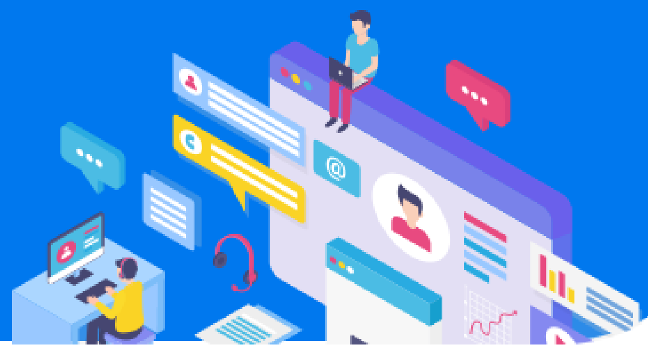
You have been assigned a task to create an Ansible playbook to automate tasks like server setup, application deployment, user management, and cloud resource provisioning.

Outcome:

By the end of this demo, you will be able to write an Ansible playbook that installs Node.js on all hosts in the **webservers** group.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Create and execute an Ansible playbook

Quick Check

Your team is adopting a new automation strategy to reduce manual intervention in repetitive tasks. Which of the following is the primary purpose of using a playbook in this context?

A. To document troubleshooting steps for end users

B. To automate and standardize tasks across multiple environments

C. To track project deadlines and milestones

D. To manage user permissions and access control

Playbook Execution in Ansible

Playbook Execution

The following concepts come into action while executing a playbook:

Sequential execution

Playbooks are read from the beginning to the end, executing tasks in the order they appear.

Multiple plays

Playbooks may have multiple plays to target different groups of hosts.

Hosts management

Within a playbook, a pattern is defined that specifies which hosts the play should run on.

Task execution

Each play in a playbook must contain at least one task to execute.

Desired State and Idempotency

Desired state

- The desired state is the specific configuration or condition that you want your managed nodes to achieve.
- For example, Ansible ensures a package is installed but will not reinstall it if that package is already present on the host machines.

Idempotency

- Idempotency refers to the property of a task or operation where running it multiple times produces the same result as running it once.
- For example, if a playbook task ensures a service is running, re-running the playbook will not restart the service.

Running a Playbook

To run the playbook, use the **ansible-playbook** command:

Example:

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

The **--verbose** flag can also be used while running a playbook to see the detailed output of both successful and unsuccessful module executions.

Running a Playbook in Check Mode

To run a playbook in check mode, you can pass the **-C** or **--check** flag with the **ansible-playbook** command:

Example:

```
ansible-playbook --check playbook.yaml
```

Ansible's check mode allows you to execute a playbook without applying any alterations to your systems. This is used to test playbooks before implementing them in a production environment.

Quick Check

Imagine you are running an Ansible playbook to configure a cluster of web servers. Which of the following statements best describes how Ansible handles the execution flow of tasks across these servers?

A. Ansible executes each task on one machine at a time before moving on to the next task.

B. Ansible executes each task on all target machines simultaneously before moving on to the next task.

C. Ansible executes all tasks on one machine before moving to the next.

D. Ansible randomly executes tasks on different machines.

Using Variables in Playbook

What Are Variables?

Variables in Ansible are used to store and reference data that can be reused throughout playbooks, roles, and tasks.

Example:

```
---
- name: Example Playbook Using Variables
  hosts: all
  vars:
    greeting: "Hello, World!"

  tasks:
    - name: Print greeting
      debug:
        msg: "{{ greeting }}"
```

The variable **greeting** is defined in the **vars** section, which holds a string value.

Best Practices for Naming Variables in Ansible

For creating variable names in Ansible, it's essential to follow certain conventions:

- Variable names should be descriptive and meaningful to convey their purpose.
- A variable name can only include letters, numbers, and underscores.
- Variable names cannot start with numbers or special characters.
- Python or playbook keywords are not valid variable names.
- Variable names are case-sensitive.

Creating a Variable Name

Examples of valid variable names:

Example:

```
basketball
basket_ball
basketball321
basket_ball321
```

✓

Example:

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

✗

Types of Variables: Simple Variables

It is a straightforward variable that stores a single value, such as a string, number, or boolean. It stores basic pieces of information that can be referenced and used throughout an Ansible playbook.

A simple variable can be defined using standard YAML syntax:

Example:

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

Referencing Simple Variables

Simple variables in Ansible can be referenced using Jinja2 template syntax, which involves wrapping the variable name in double curly braces {{ }}.

Example:

```
ansible.builtin.template:
  src: foo.cfg.j2
  dest: '{{ remote_install_path }}/foo.cfg'
```

In this example, the variable **remote_install_path** defines the location of a file where the **ansible.builtin.template** template will be placed.

Types of Variables: List Variables

A list variable combines a variable name with multiple values. These multiple values can be stored as an itemized list or in square brackets [], separated with commas.

They can be defined using standard YAML syntax.

Example:

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

Referencing List Variables

Items in a list variable are indexed by default, which means the first item in a list is item 0, the second item is item 1, and so on. Therefore, list items can be referenced using their indices.

Example:

```
region: "{{ region[0] }}"
```

In this example, the first item of the list variable is being referenced.

Types of Variables: Dictionary Variables

A dictionary stores the data in key-value pairs. Usually, dictionaries are used to store related data, such as the information contained in an ID or a user profile.

Dictionary variable in Ansible are defined as a key-pair value:

Example:

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

In the example, a dictionary variable **foo** is defined with two key-value pairs: **field1** set to **one** and **field2** set to **two**.

Referencing Dictionary Variables

To access the values stored in the dictionary, you can use either dot notation or bracket notation.

Example:

```
foo['field1']
foo.field1
```

This approach allows organized and readable access to complex data structures within an Ansible playbook.

Defining Variables in Inventory

Defining variables in an inventory file allows you to set values that are applied to specific hosts or groups of hosts.

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.

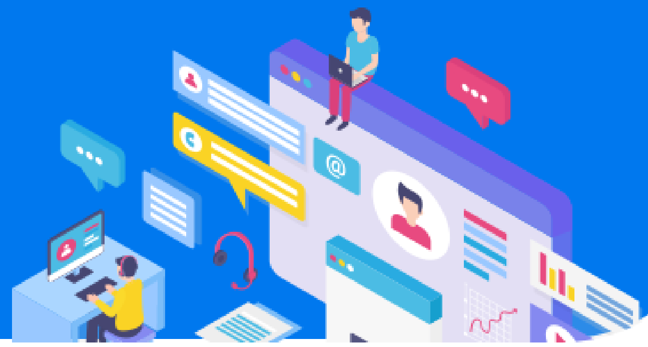
Defining Variables in Inventory

The following is an example of how to define variables in an Ansible inventory:

Example:

```
all:
  hosts:
    server1:
    server2:
  vars:
    variable_name: value
```

Assisted Practice



Executing variables in Ansible playbook **Duration: 15 Min.**

Problem statement:

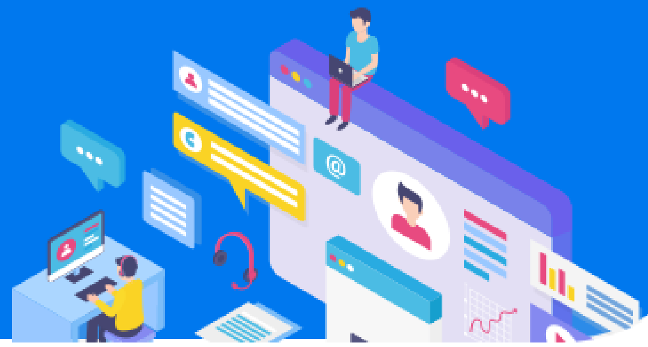
You have been assigned a task to execute variables in the Ansible playbook to manage dynamic configurations and settings.

Outcome:

By the end of this demo, you will be able to use variables in your Ansible playbook to define values that can change depending on the context in which you run the playbook.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Execute variables in the Ansible playbook

Quick Check

In an Ansible playbook, which of the following is the correct way to reference a variable defined as a dictionary in your inventory file?

A. {{ inventory['variable_name'] }}

B. {{ variable_name.key }}

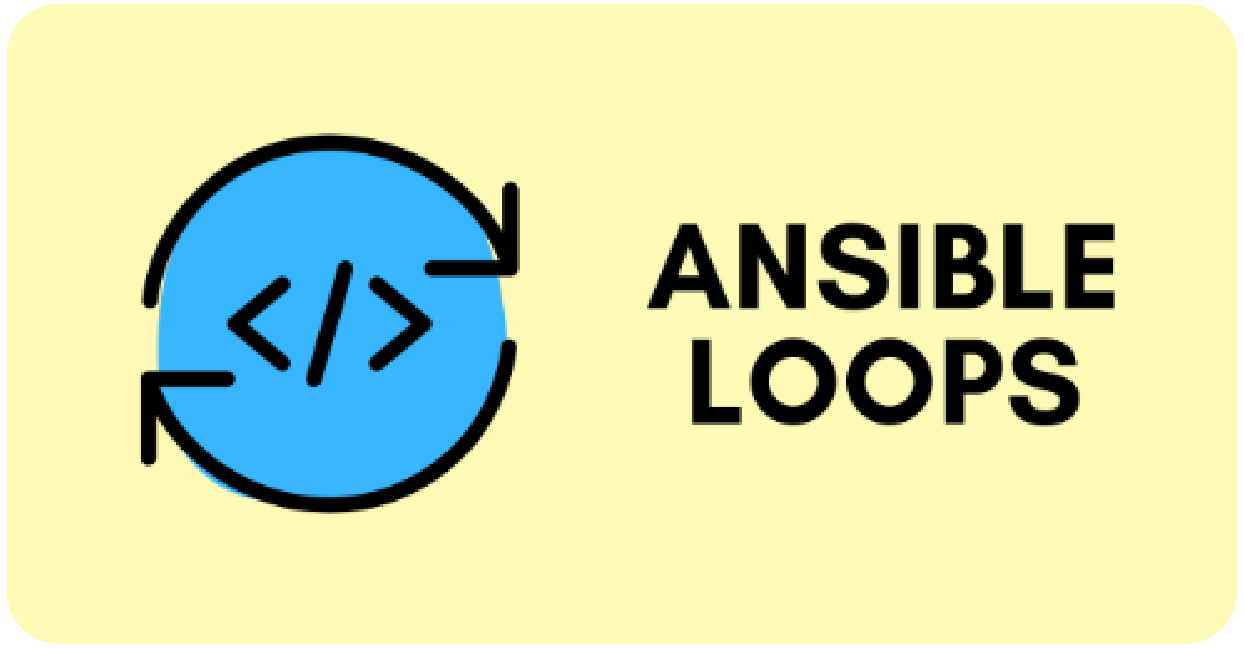
C. {{ variable_name['key'] }}

D. {{ key.variable_name }}

Task Iterations with Loops

Ansible Loops

In Ansible, loops are used to iterate over lists, dictionaries, or other iterable data structures to perform repetitive tasks efficiently.



Ansible offers the **loop**, **with_<lookup>**, and **until** keywords to execute a task multiple times.

Use Cases of Ansible Loops

Installing multiple packages	Loops can simplify and improve maintainability when installing a list of server packages.
Managing users	Loops can streamline the process of creating or managing multiple user accounts.
Creating directories	Loops can efficiently handle the creation of directory structures for projects or applications.

Types of Loops in Ansible

There are several types of loops available in Ansible, each suited for different use cases:

1. Standard loops
2. Nested loops
3. Looping over range
4. Dictionary loops

Standard Loops

with_items: This is the most common loop in Ansible. It iterates over a list of items and executes tasks for each item.

Example:

```
- name: Example of with_items
  debug:
    msg: "Item: {{ item }}"
  with_items:
    - apple
    - banana
    - orange
```

Each item is accessed using the **item** variable within the loop.

In this example, the **debug** module will print each item in the list (**apple**, **banana**, **orange**).

Standard Loops

loop: While **with_items** is widely used, Ansible encourages the use of the **loop** keyword as a more modern and versatile alternative.

Example:

```
- name: Example of loop
  debug:
    msg: "Item: {{ item }}"
  loop:
    - apple
    - banana
    - orange
```

The **loop** directive indicates that the task should be repeated for each item in the list.

Nested Loops

Nested loops in Ansible allow you to iterate over multiple lists simultaneously, which is particularly useful when you need to perform actions that involve combinations of elements from different lists.

Example:

```
- name: Example of nested loop
  debug:
    msg: "Item1: {{ item1 }},
Item2: {{ item2 }}"
  with_nested:
    - [1, 2, 3]
    - ['a', 'b', 'c']
```

The **with_nested** directive will iterate over the provided list of lists.

Looping over Range

In Ansible, looping over a range lets you perform a task for a specified number of times or iterate over a series of numbers. This can be done by using both the **with_sequence** and **loop** keywords.

The following is an example of looping over a range using **with_sequence** and **loop**:

Example:

```
- name: Example of with_sequence
  debug:
    msg: "{{ item }}"
  with_sequence: start=0 end=4
stride=2 format=testuser%02x
```

Example:

```
- name: Example of loop
  debug:
    msg: "{{ 'testuser%02x' |
format(item) }}"
  # range is exclusive of the endpoint
  loop: "{{ range(0, 4 + 1, 2)|list }}"
```

Dictionary Loops

In Ansible, you can loop over dictionaries to perform tasks on each key-value pair. This can be achieved using the **with_dict** keyword or the more modern **loop** keyword combined with the **dict2items** filter.

The following is an example of a dictionary loop using **with_dict** and **loop**:

Example:

```
- name: with_dict
  debug:
    msg: "{{ item.key }} - {{
item.value }}"
  with_dict: "{{ dictionary }}"
```

Example:

```
- name: loop
  debug:
    msg: "{{ item.key }} - {{ item.value
}}"
  loop: "{{ dictionary|dict2items }}"
```

Using the Until Condition

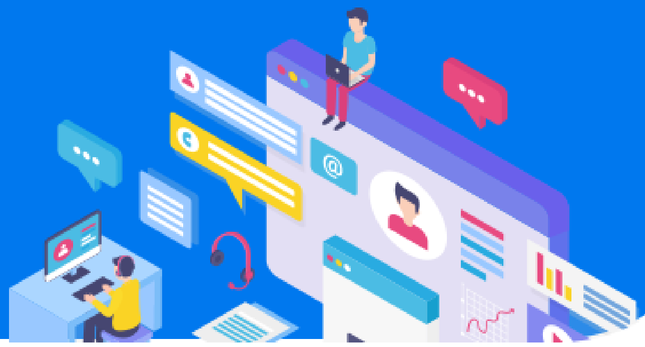
You can use the **until** keyword to retry a task until a certain condition is met. Here’s an example:

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
```

The above example runs the shell module recursively until the module’s result has **all systems go** in its **stdout** or the task has been retried 5 times with a delay of 10 seconds.

Assisted Practice



Executing loops in playbook **Duration: 15 Min.**

Problem statement:

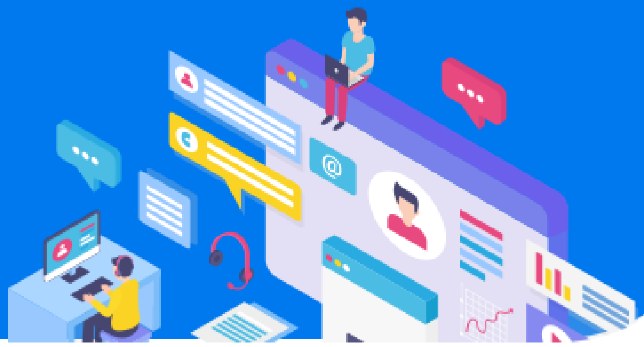
You have been assigned a task to execute loops in a playbook to iterate over a list of items efficiently in Ansible.

Outcome:

By the end of this demo, you will be able to use loops in Ansible playbooks to perform repetitive tasks efficiently, allowing you to apply the same task to multiple items or hosts without duplicating code, thereby simplifying playbook management and improving readability.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Create and execute a playbook

Quick Check

You are using Ansible to install a list of packages on multiple servers. If the package installation fails, it should be retried up to three times. Which Ansible task configuration correctly implements this requirement?

A. Use **with_items** to loop over packages with retry logic

B. Use **loop** to iterate over packages and include retry logic

C. Use **with_dict** to loop over servers and loop for packages with retry logic

D. Use **with_list** to loop over packages and include retries

Conditional Tasks with Ansible

Conditional Tasks in Ansible Playbook

It provides a powerful mechanism to control the execution of tasks based on specific conditions.

These conditions can be related to the state of the target hosts, the results of previous tasks, or the values of variables.

Conditional Tasks in Ansible Playbook

The following are some uses of conditional tasks in Ansible playbooks:

01 **Flexibility**

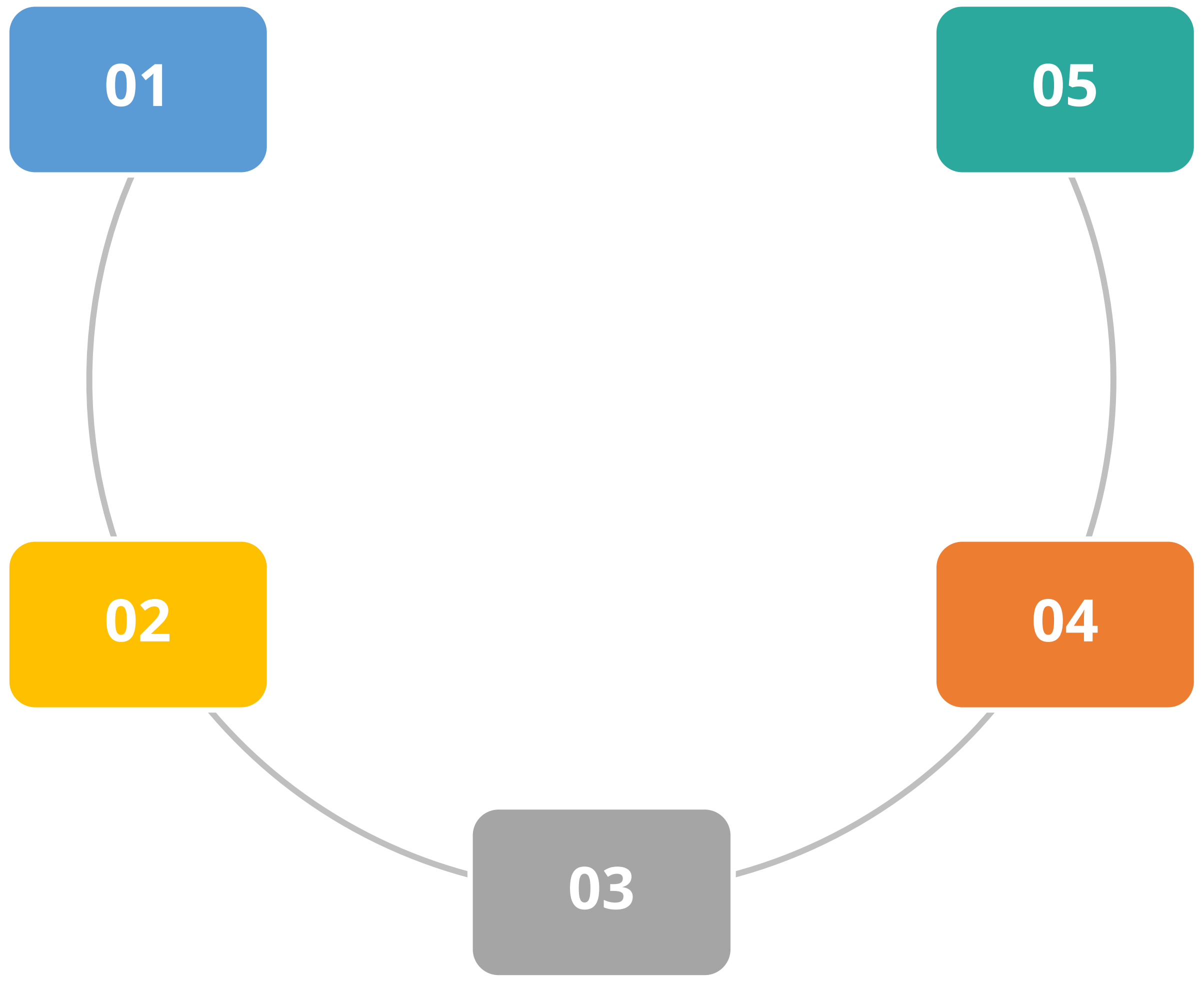
Conditional tasks allow you to tailor playbook execution based on dynamic conditions, making your automation more flexible and adaptable to different environments.

02 **Error handling**

Conditional tasks let you build error-handling logic, helping you manage failures smoothly and adapt to unexpected issues during playbook execution.

Conditional Tasks in Ansible Playbook

The most common ways of implementing Ansible conditionals are:



When statement

Conditionals with loops

Custom facts

Register variables

Conditional imports

When Statement

The **when** conditional in Ansible is used to control the execution of tasks based on the evaluation of a condition. It uses Jinja2 expressions to evaluate conditions. Here’s a simple example:

Example:

```
- hosts: all
  tasks:
    - name: Install httpd when on a RedHat-based system
      yum:
        name: httpd
        state: present
      when: ansible_os_family == "RedHat"
```

Using Variables in Conditionals

The following example shows the use of variables with the **when** conditional to execute tasks based on dynamic data:

```
- hosts: all
  vars:
    http_port: 80
  tasks:
    - name: Open HTTP port on firewall
      firewalld:
        port: "{{ http_port }}/tcp"
        permanent: yes
        state: enabled
      when: http_port == 80
```

Note:

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

Conditionals with Loop

Conditional statements with loops in Ansible can be highly effective for controlling task execution based on dynamic conditions.

Example:

```yaml
- hosts: all
  tasks:
    - name: Ensure services are started
      service:
        name: "{{ item }}"
        state: started
      loop:
        - httpd
        - nginx
      when:
        - ansible_os_family == "RedHat"
        - "'{{ item }}' not in ansible_facts.services"
```

Combining Conditions

The following example shows how to combine multiple conditions using **and**, **or**, and parentheses to control the logical flow:

```
- hosts: all
  tasks:
    - name: Install packages based on multiple conditions
      yum:
        name: "{{ item }}"
        state: present
      loop:
        - httpd
        - nginx
      when: ansible_os_family == "RedHat" and
ansible_distribution_major_version | int >= 7
```

Conditionals with Custom Facts

Ansible custom facts allow users to define additional variables beyond Ansible's default facts. These facts can then be used in conditional statements to control the execution of tasks.

Example:

```
- hosts: all
  tasks:
    - name: Create custom fact file
      copy:
        dest: /etc/ansible/facts.d/service_status.fact
        content: |
          {
            "service_enabled": "yes"
          }
        mode: '0644'

    - name: Gather facts
      setup:

    - name: Start httpd service if enabled
      service:
        name: httpd
        state: started
      when: ansible_local.service_status.service_enabled == "yes"
```

Conditional Imports

Ansible allows for conditional imports to dynamically include tasks, roles, or playbooks based on specific conditions. The following is an example of importing tasks using conditionals:

Example:

```
- hosts: all
  tasks:
    - name: Check if the operating system is RedHat
      set_fact:
        is_redhat: "{{ ansible_os_family == 'RedHat' }}"

    - name: Import tasks for RedHat systems
      import_tasks: redhat_tasks.yml
      when: is_redhat

    - name: Import tasks for other systems
      import_tasks: other_tasks.yml
      when: not is_redhat
```

Conditionals with Register Variables

In Ansible, variables can be registered to capture the output of tasks, which can then be used in conditional statements to control the execution of subsequent tasks.

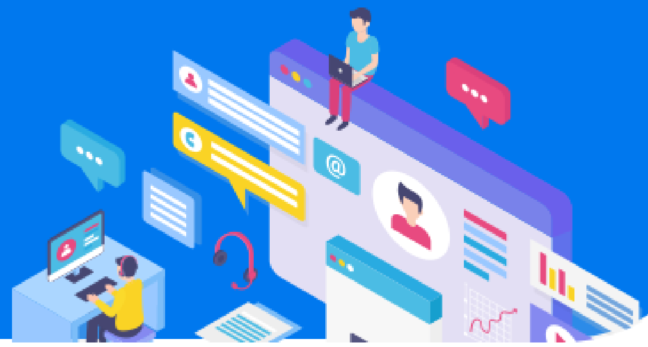
Example:

```
- hosts: all
  tasks:
    - name: Check if the file exists
      stat:
        path: /path/to/file
      register: file_status

    - name: Create the file if it does not exist
      file:
        path: /path/to/file
        state: touch
      when: not file_status.stat.exists

    - name: Notify if the file exists
      debug:
        msg: "The file already exists."
      when: file_status.stat.exists
```

Assisted Practice



Implementing conditionals in playbooks **Duration: 15 Min.**

Problem statement:

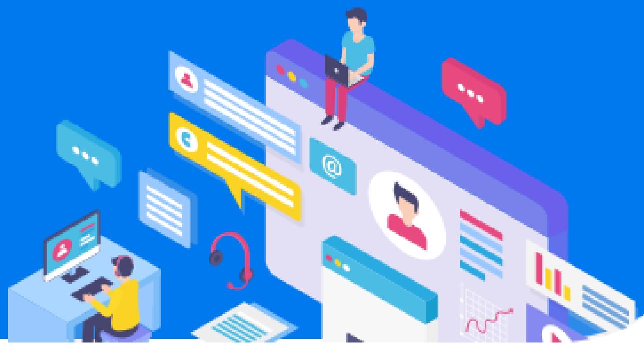
You have been assigned a task to implement conditionals in playbooks to control the execution of tasks based on specific conditions.

Outcome:

By the end of this demo, you will be able to use conditionals in Ansible playbooks to execute tasks selectively based on specific criteria or conditions, allowing for more dynamic and flexible automation by tailoring actions to different situations and environments.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Create and run the playbook containing conditionals

Quick Check

You are using Ansible to configure servers. A task should run only if the variable **deploy** is set to **yes** and the server's custom fact environment is set to **production**. Which Ansible task configuration correctly implements these conditions?

A. Use **when: deploy and environment == 'production'**

B. Use **when: deploy == 'yes' and environment == 'production'**

C. Use **when: deploy is yes and environment is production**

D. Use **when: deploy == 'yes' or environment == 'production'**

Ansible Handlers

What Are Ansible Handlers?

Ansible handlers are special tasks that are triggered by other tasks to run only when notified. They are typically used for tasks that need to be performed only if there is a change in the system.

For example, if a configuration file is updated, a handler can be notified to restart the service to apply the new configuration.

Example of a Playbook with Handler

```
- hosts: all

  tasks:
    - name: Install Apache
      yum:
        name: httpd
        state: present
      notify: Restart Apache

    - name: Update Apache configuration
      copy:
        src: /path/to/httpd.conf
        dest: /etc/httpd/conf/httpd.conf
      notify: Restart Apache

  handlers:
    - name: Restart Apache
      service:
        name: httpd
        state: restarted
```

Controlling When Handlers Run

By default, Ansible handlers run after all tasks in a play are completed. To run handlers before the end of the play, use the **meta: flush_handlers** task, which triggers any notified handlers at that point in the play.

Example:

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

  - name: Flush handlers
    meta: flush_handlers

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

It is useful for scenarios where immediate feedback is necessary, such as restarting a service right after updating its configuration, without waiting for all other tasks to complete.

Variables with Handlers

In Ansible, handlers can also use variables to perform dynamic actions based on the context in which they are executed. This allows for greater flexibility and reusability of handlers.

Example:

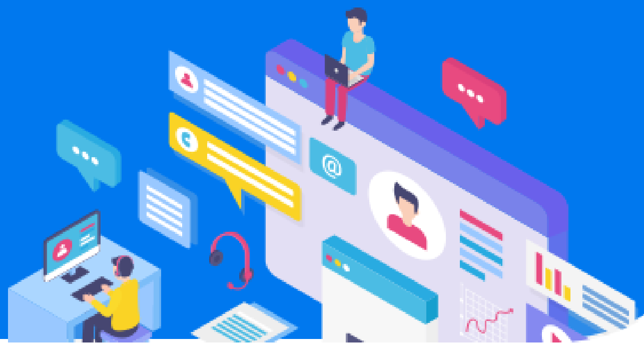
```
- hosts: all
  vars:
    service_name: nginx
    config_file: /etc/nginx/nginx.conf

  tasks:
    - name: Install NGINX
      apt:
        name: "{{ service_name }}"
        state: present
      notify: Restart NGINX

    - name: Update NGINX configuration
      copy:
        src: /path/to/nginx.conf
        dest: "{{ config_file }}"
      notify: Restart NGINX

  handlers:
    - name: Restart NGINX
      service:
        name: "{{ service_name }}"
        state: restarted
```

Assisted Practice



Configuring tasks with handlers **Duration: 20 Min.**

Problem statement:

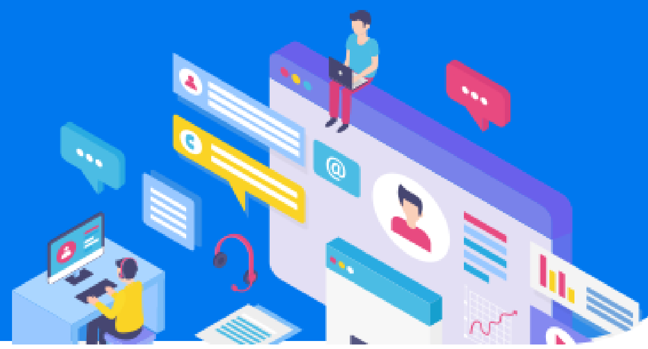
You have been assigned a task to configure an Ansible playbook to execute tasks on localhost and trigger handlers for subsequent actions upon task completion.

Outcome:

By the end of this demo, you will be able to use handlers in Ansible playbooks to execute tasks only when notified by other tasks, such as restarting a service after a configuration file is updated, ensuring efficient and conditional execution.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Create the playbook and inventory file
2. Run the playbook

Quick Check

You are using Ansible to manage a web server. You have multiple tasks that notify a handler to restart the web service. You want to ensure that the handler runs immediately after a specific task rather than waiting until the end of the play. Which Ansible option allows you to do so?

A. flush_handlers

B. notify_now

C. run_immediately

D. immediate_notify

Ansible Filters

What Are Ansible Filters?

Ansible filters are tools used within Jinja2 templates to transform data. They allow you to manipulate variables and strings, perform calculations, and format data.

- Filters can change the format or content of data, such as converting a string to uppercase.
- Filters can perform mathematical operations and logical comparisons, enabling complex data manipulation.
- Filters are built on Jinja2 templating, which is used for variable interpolation and template rendering.

Commonly Used Filters

String formatting filters	• **lower**: Converts a string to lowercase • **upper**: Converts a string to uppercase • **replace**: Replaces occurrences of a substring with another string
Number formatting filters	• **round:** Rounds a number to a specified number of decimal places • **abs:** Returns the absolute value of a number • **int:** Converts a value to an integer

Commonly Used Filters

Date and time formatting filters	• **strftime:** Formats a date string according to a specified format • **timestamp:** Converts a date string to a Unix timestamp
List and dictionary formatting filters	• **sort:** Sorts a list • **unique:** Removes duplicate items from a list • **join:** Joins list items into a single string with a specified separator

Filters for Data Structures

The following filters will take a data structure in a template and render it in a slightly different format:

To convert data format

```
{{ some_variable | to_json }}
{{ some_variable | to_yaml }}
```

To convert into human-readable format

```
{{ some_variable | to_nice_json }}
{{ some_variable | to_nice_yaml }}
```

To parse from a formatted string

```
{{ some_variable | from_json }}
{{ some_variable | from_yaml }}
```

List Filters

The following filters operate on list variables:

To get the minimum value	To get the maximum value	To flatten a list
{{ list1 | min }}	**{{ [3, 4, 2] | max }}**	**{{ [3, [4, 2]] | flatten }}**

Dictionary Filters

The following filters operate on dictionary variables:

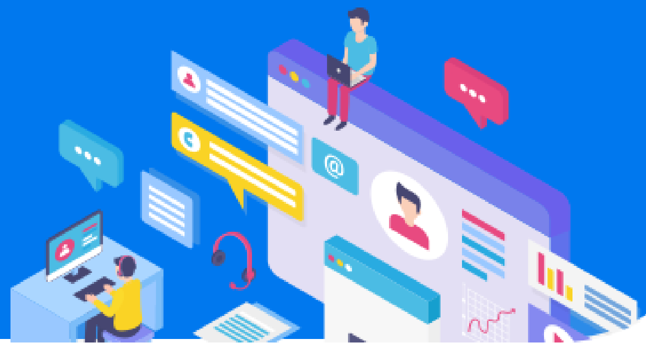
To turn a dictionary into a list

```
{{ dict | dict2items }}
```

To turn a list of dicts with keys into a dict

```
{{ tags | items2dict }}
```

Assisted Practice



Working with Ansible filters **Duration: 20 Min.**

Problem statement:

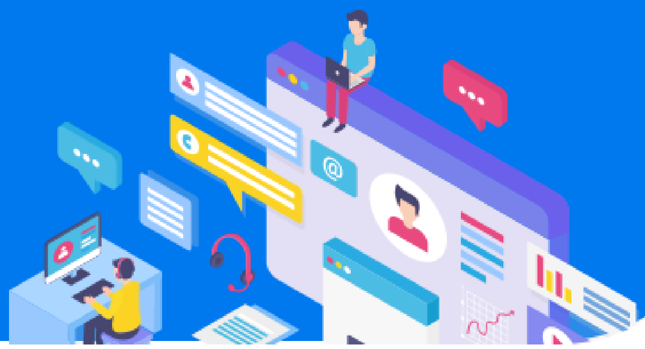
You have been assigned a task to work with predefined and custom Ansible filters for manipulating data and transforming it into the desired format.

Outcome:

By the end of this demo, you will be able to use filters in Ansible playbooks to manipulate and transform data within tasks, allowing for more precise control over variables and output, such as formatting strings, modifying lists, or performing calculations, to meet specific requirements.

Note: Refer to the demo document for detailed steps

Assisted Practice: Guidelines



Steps to be followed:

1. Create the initial directory structure
2. Create a playbook to use predefined Ansible filters
3. Create a playbook to use custom Ansible filters
4. Run the playbooks

Quick Check

You are using Ansible to manage configurations and need to format strings in your playbooks. Specifically, you need to convert a hostname to uppercase and ensure that a certain string is included in all uppercase letters. Which Ansible filter would you use to convert a string to uppercase?

A. upper

B. uppercase

C. to_upper

D. capitalize

Key Takeaways

- An Ansible playbook is a YAML (Yet Another Markup Language) script that defines tasks for remote hosts, enabling the automation of complex configurations in a human-readable format.
- Tasks in Ansible playbooks define the actions to be performed on the target hosts.
- Variables in Ansible are used to store and reference data that can be reused throughout playbooks, roles, and tasks.
- Ansible offers the **loop**, **with_<lookup>**, and **until** keywords to execute a task multiple times.
- A simple variable is a variable name with a single value. It is useful for storing singular pieces of information in a playbook.

Key Takeaways

- A list variable combines a variable name with multiple values. These values can be stored as an itemized list or in square brackets [], separated with commas.
- A dictionary variable, also known as a hash or map, is a data structure that stores key-value pairs.
- Conditional tasks in Ansible provide a powerful mechanism to control the execution of tasks based on specific conditions.
- Ansible handlers are special tasks triggered by other tasks to run only when notified.
- Ansible filters allow users to manipulate variables and strings, perform calculations, and format data.

Installing NodeJS Using Ansible Playbook

Duration: 25 Min.

Project agenda: To install NodeJS using an Ansible playbook for web development environments

Description: You work as a junior DevOps engineer in an IT firm. Your company is undertaking a project that involves setting up multiple web development environments. One of the key requirements is to automate the installation of NodeJS across all servers using Ansible to ensure consistency and efficiency.

Installing NodeJS Using Ansible Playbook

Duration: 25 Min.

Perform the following:

1. Install Ansible
2. Configure Ansible
3. Establish connectivity between the Ansible controller and node machine
4. Create a NodeJS playbook
5. Execute the playbook
6. Confirm the installation

Expected deliverables: A playbook to install NodeJS and check the installation status of NodeJS.

Thank you