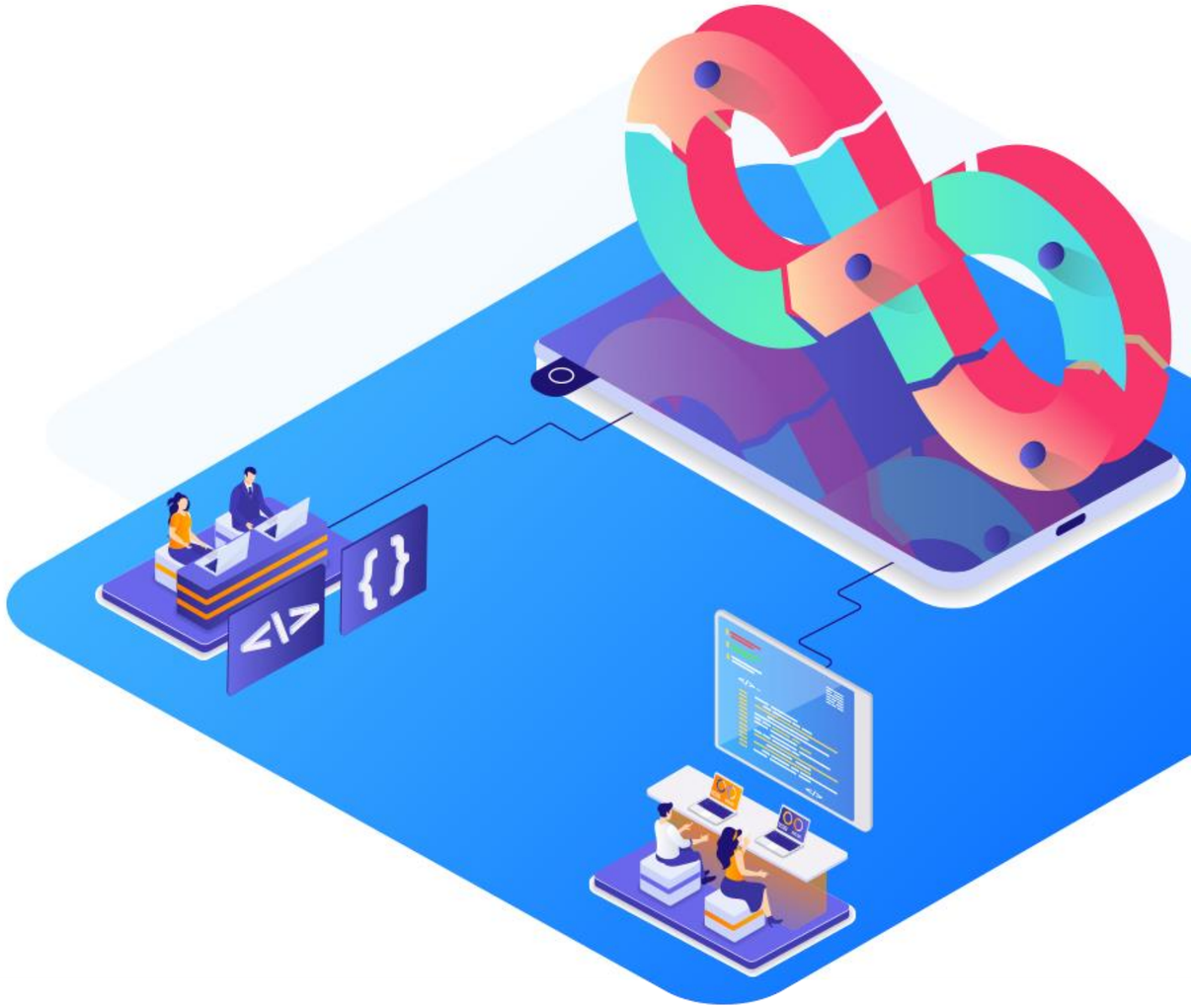
Configuration Management with Ansible and Terraform



Ansible Jinja2 Templates

Learning Objectives

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

- Describe the need for a Jinja2 template in streamlining configuration management
- Illustrate the role of variables in storing program and manipulating data
- Analyze the effects of using multiple filters together for data processing
- Demonstrate conditional statements with arrays in a Jinja2 template for dynamically rendering content based on the array's elements

Learning Objectives

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

- Implement loops within Jinja2 templates for rendering lists or arrays
- Illustrate the outcome of a test in a template based on certain conditions

Introduction to Jinja2 Template

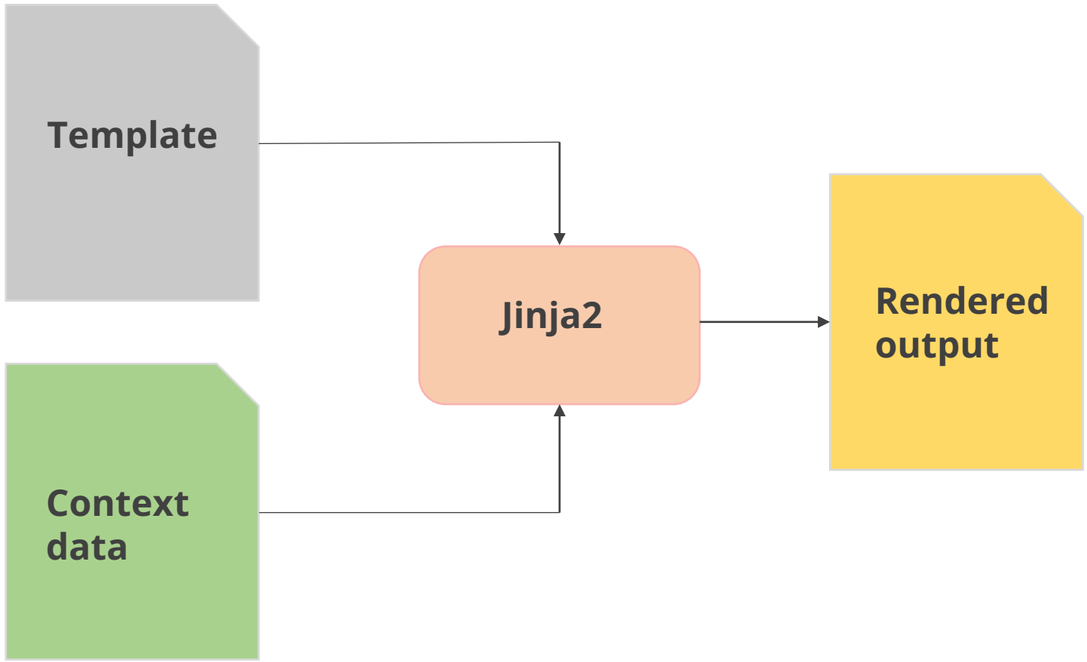
What Is Jinja2?

It is a popular templating language known for its user-friendly design that allows dynamic content generation by embedding expressions within templates.

It is used in various applications such as web frameworks like Flask and Django, configuration management tools like Ansible, and static site generators to render dynamic content.

Basic Workflow of Jinja2

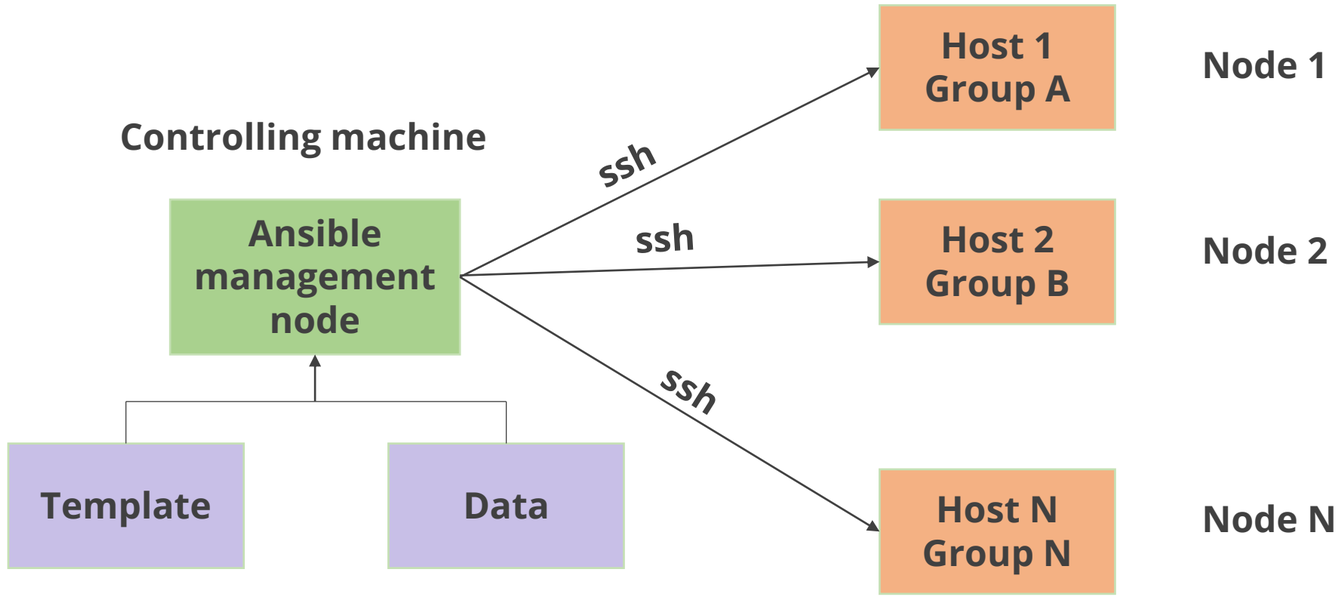
Jinja2 renders the final document using two main components: **templates** (predefined structures) and **dynamic context data** (JSON).



The Jinja2 engine processes the template with the provided context data, producing a customized and dynamic output.

Where Is Jinja2 Templating Done?

Jinja2 templates are processed on the Ansible controller machine, where the templates and data are combined to generate the final task configurations.



These configurations are then executed on the target machines (nodes) via SSH connections, ensuring that each host receives the appropriate settings based on the templated data.

Jinja2: Delimiters

They are the special markers used to denote the start and end of code segments, such as variables, control statements, comments, and line statements within templates.

The default Jinja2 delimiters are configured as follows:

{{ }}	These are used for embedding variables which print their actual value during code execution.
{% %}	These are used for control statements such as loops and if-else.
{# #}	These are used to specify comments.
# .. ##	These are used to specify line statements.

Jinja2: Example

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Jinja2 Short Example</title>
</head>
<body>
    <!-- Using {{ }} for variable substitution -->
    <h1>Hello, {{ name }}!</h1>

    <!-- Using {% %} for control structures -->
    {% if is_logged_in %}
        <p>Welcome back, {{ name }}!</p>
    {% endif %}

    <!-- Using {% %} for loops -->
    <ul>
    {% for item in items %}
        <li>{{ item }}</li>
    {% endfor %}
    </ul>

    <!-- Using {# #} for comments -->
    {# This is a comment and won't be rendered in the final
output #}
</body>
```

When to Use Jinja2?

Below are examples showcasing various scenarios where Jinja2 can be effectively used:

01 To create dynamic web content by generating HTML pages with content that changes based on user inputs or database queries

02 To generate personalized email templates tailored to each recipient's information

03 To build automated configuration files for applications or servers with varying inputs

04 To produce reports with dynamic data that can be exported as HTML, PDF, or other formats

Where Is Jinja2 Used?

Below is the detailed overview of its applications:

Web frameworks

Generate dynamic HTML pages in web frameworks like Flask and Bottle

Configuration management

Configure files in Ansible for managing large IT systems

Static site generators

Create static websites using generators like Pelican

Security and vulnerabilities

Address security vulnerabilities, such as the recent CVE-2024-22195

DevOps and monitoring tools

Customize alert messages in Grafana from JSON payloads

AI and chatbots

Format prompts for chatbots and AI applications

Assisted Practice

Setting up Jinja2 **Duration: 15 Min.**

Problem statement:

You've been assigned a task to install and set up Jinja2 to create dynamic and flexible templates.

Outcome:

You can successfully install and set up Jinja2 and create dynamic and flexible templates in your Python applications.

Note: Refer to the demo document for detailed steps:
01_Setting_up_Jinja2

Assisted Practice: Guidelines

Steps to be followed:

1. Install Jinja2
2. Verify the installation of Jinja2

ASSISTED PRACTICE

Quick Check

You are working on an automation project where you need to generate configuration files dynamically based on various input parameters. Which tool would you use to create a template to handle this task efficiently, and why?

A. XML

B. YAML

C. Jinja2

D. JSON

Variables in Jinja2

What Is Variable?

It is a placeholder used to store and manage dynamic data within a template.

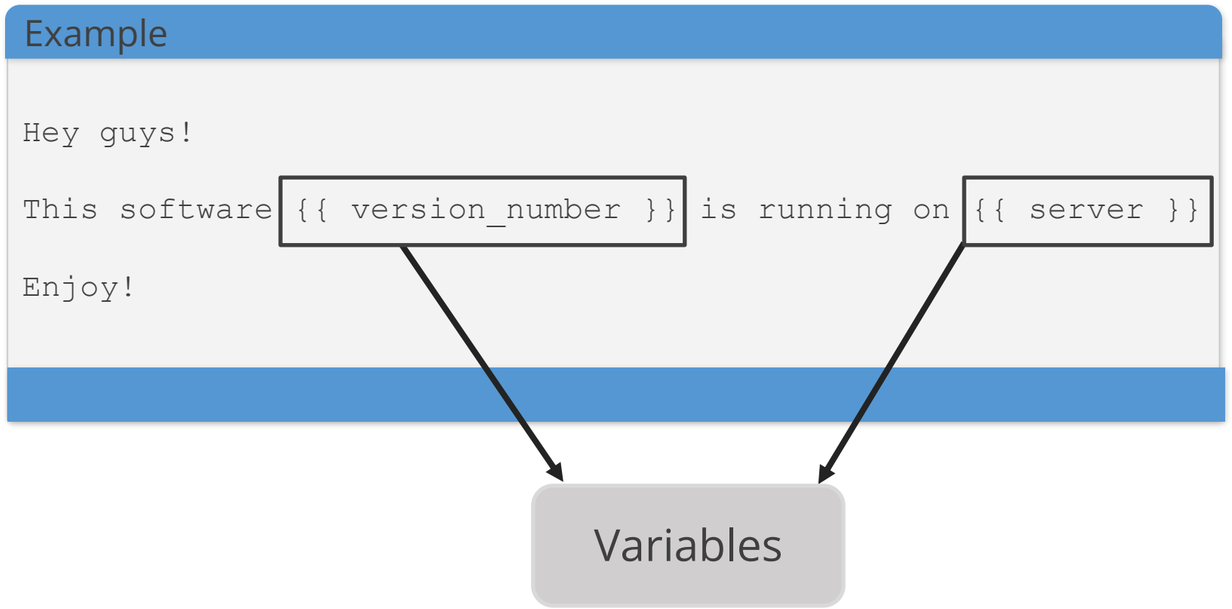
Syntax

```
{{ variable_name }}
```

It allows you to insert values, such as numbers, strings, lists, dictionaries, or objects, into the template.

How Variables Work?

When rendering a template, Jinja2 evaluates and replaces the variables with the actual data provided.



This process involves taking the template file with placeholders and substituting these placeholders with the corresponding values from the context data passed to the template.

Variables in Templates

The variables can be used within templates to dynamically render content. This allows for flexible and responsive template design based on varying data inputs.

Python

```
from jinja2 import Template

# Define the template with a variable
template = Template("Hello, {{ name }}!")

# Render the template with a variable value
rendered = template.render(name="Alice")

print(rendered)
```

Output

```
Hello, Alice!
```

In the above example, the variable name is defined in the Python code and passed to the Jinja2 template, which then renders content dynamically.

Variables: Uses

Below are some common uses of variables within Jinja2 templates:

Simple variables: They are represented using double curly braces {{ }}.

Example

```
<p>{{ user.name }}</p>
<p>{{ user.age }}</p>
```

Filters: They are appended to the variable with a pipe | symbol.

Example

```
<p>{{ user.name|upper }}</p>
 <!-- Converts the name to uppercase -->
<p>{{ user.age|default(25) }}</p>
<!-- Sets a default value if the age is
not provided -->
```

Variables: Uses

Below are some common uses of variables within Jinja2 templates:

Conditional statements: They use {% if %} and {% endif %} tags to denote the beginning and end of the conditional block.

Example

```
{% if user.age >= 18 %}
  <p>Welcome, {{ user.name }}!</p>
{% else %}
  <p>Sorry, {{ user.name }}. You must be
at least 18 years old.</p>
{% endif %}
```

Loops: They are defined using {% for %} and {% endfor %} tags.

Example

```
<ul>
  {% for item in items %}
    <li>{{ item }}</li>
  {% endfor %}
</ul>
```

Assisted Practice

Accessing variables in Jinja2 **Duration: 15 Min.**

Problem statement:

You've been assigned a task to access variables in Jinja2 for dynamic template rendering.

Outcome:

You can successfully access variables in Jinja2 for dynamic web content generation, enabling more interactive and personalized user experience.

Note: Refer to the demo document for detailed steps:
02_Accessing_Variables_in_Jinja2

Assisted Practice: Guidelines

Steps to be followed:

1. Create a file and define the template
2. Load the Python script

Quick Check

You are working on an Ansible playbook that needs to deploy different applications to various servers. Each application requires a specific port number that is different for each server. How would you use a variable to handle this situation?

A. Create a separate playbook for each server with hardcoded port numbers

B. Define a variable for the port number in the variable files and use it in your Jinja2 template

C. Use a control structure to loop through each port number

D. Use template inheritance to define the port number

Filters in Jinja2

What Is Filter?

It is applied to variable names using the pipe (**|**) syntax, allowing for modifying output appearance or data formatting.

Syntax

```
{{ variable | argument }}
```

It enhances template rendering by enabling operations like string manipulation, date formatting, and data transformations.

Chained Filters

It refers to applying multiple filters in sequence to a variable or value. Each filter processes the value and passes its output to the next filter.

Syntax

```
{{ name|title|subtitle }}
```

Filters can be chained if each filter's output type matches the input type expected by the next filter.

Chained Filters: Example

Filters are evaluated from left to right.

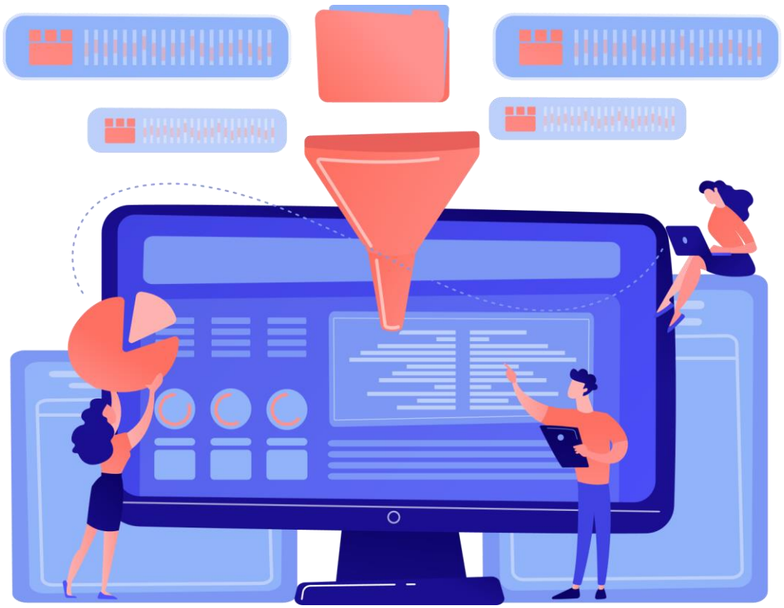
Example

```
{{ "example" | upper | length }}
```

- The upper filter converts **example** to **EXAMPLE**.
- The length filter counts the number of characters in **EXAMPLE**, resulting in **7**.

Default Filter

It prevents errors by assigning a fallback value to undefined variables during template rendering, ensuring smoother playbook execution.



In Ansible, it enhances playbook reliability by robustly managing potential variable absence, fostering resilient infrastructure management.

Default Filter of Jinja2

The **default** filter can be used to provide default values for variables directly in the templates.

Syntax

```
{{ missing_variable | default(2) }}
```

Variables can be made optional and mandatory using the keywords **omit** and **mandatory,** respectively.

Format Filter

It allows for string formatting in Jinja2 templates. It can take a sequence of values (args) or keyword arguments (kwargs) and apply them to a string template.

Syntax

```
{{ 'string_template' | format(args) }}
{{ 'string_template' | format(kwargs) }}
```

Note:

Jinja2 string formatting does not support formatting variables.
Use **variable=value** keyword arguments instead.

Format Filter: Example

Args: A sequence of values assigned to the % operator sequentially

Example

```
{{ '%s after %s, %d by %d' | format('One', 'another', 1,
1) }}
// Output: 'One after another, 1 by 1'
```

Kwargs: A sequence of keyword arguments in the form keyword=value

Example

```
{{ 'These %(items)s have %(verb)s argument
order.'|format(verb='swapped', items='words') }}
// Output: 'These words have swapped argument order.'
```

Filters and Arguments

01 The filters act like functions and require at least one argument.

02 Place the first argument before the pipe symbol.

03 Place extra arguments in round brackets after the filter name. Round brackets are optional if no extra arguments are needed.

Filters and Arguments: Example

Filter without extra arguments:

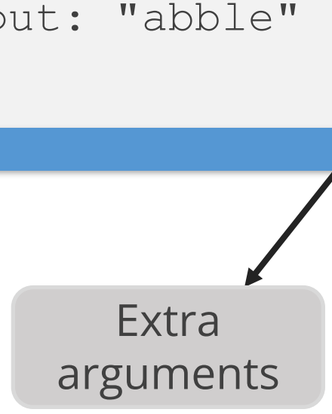
Example

```
{{ "orange" | upper }}
{{ "orange" | upper() }}
```

Filter with extra arguments:

Example

```
{{ "apple" | replace }}
// Error: Missing arguments
{{ "apple" | replace('p', 'b') }}
// Output: "abble"
```



Built-in Filters

Following are some of the most used built-in filters:

abs()	float()	lower()	round()	tojson()
attr()	forceescape()	map()	safe()	trim()
batch()	format()	max()	select()	truncate()
capitalize()	groupby()	min()	selectattr()	unique()
center()	indent()	pprint()	slice()	upper()
default()	int()	random()	sort()	urlencode()
dictsort()	join()	reject()	string()	urlize()
escape()	last()	rejectattr()	striptags()	wordcount()
filesizeformat()	length()	replace()	sum()	wordwrap()
first()	list()	reverse()	title()	xmlattr()

Assisted Practice

Implementing variables with Jinja2 filters **Duration: 15 Min.**

Problem statement:
You have been assigned the task of processing variables in Jinja2 filters for enhancing data manipulation and formatting capabilities within template rendering processes.

Outcome:
You can successfully enhance data manipulation and formatting within Jinja2 templates using filters, resulting in more dynamic and customized content presentations tailored to specific requirements, thereby optimizing your development workflow.

Note: Refer to the demo document for detailed steps:
03_Implementing_Variables_in_Jinja2_Filters

ASSISTED PRACTICE

Assisted Practice: Guidelines

Steps to be followed:

1. Create a playbook and configure it using Jinja2
2. Create a playbook and configure it using default filters

ASSISTED PRACTICE

Quick Check

You need to apply multiple transformations to a variable in a Jinja2 template, such as converting text to uppercase and trimming whitespace. Which feature of Jinja2 allows you to achieve this?

A. Template inheritance

B. Chained filters

C. Macros

D. Control structures

Arrays

What Is an Array?

It is a data structure that stores multiple values in a single variable, represented as lists.

Syntax

```
{% set arrayName = [value1, value2, value3] %}
```

It allows you to manage and access a collection of items within the templates efficiently.

Arrays: Example

Example

```
{% set fruits = ['apple', 'banana', 'cherry'] %}
    {{ fruits[0] }}  <!-- Output: apple -->
```

- The array **fruits** is initialized with three values: 'apple', 'banana', and 'cherry'.
- Using {{ fruits[0] }} accesses the first element of the array, that is **apple**.

Conditional Statements with Arrays

Jinja2 uses **if** statements to perform conditional checks on arrays.

Syntax

```
{% if condition %}
  {{ value_if_true }}
{% endif %}
```

Example

```
{% if 'red' in colors %}
  {{ 'Red is in the list!' }}
{% endif %}
{# Outputs: 'Red is in the list!' #}
```

Looping through Arrays

Jinja2 uses the **for** loop to iterate over elements in an array.

Syntax

```
{% for item in array_name %}
  {{ item }}
{% endfor %}
```

Example

```
{% for color in colors %}
  {{ color }}
{% endfor %}
{# Outputs: 'red', 'green', 'blue' #}
```

Array Filters

Filters like length, sort, and unique can be used to manipulate arrays.

Syntax

```
{{ array_name | filter_name }}
```

Example

```
{{ colors | length }}  {# Outputs: 3 #}
{{ colors | sort }}    {# Outputs: ['blue',
'green', 'red'] #}
```

Quick Check

You have an array users = ['admin', 'guest', 'user'] in your Jinja2 template. How would you write a conditional statement to check if 'admin' is in the array?

A. {% if 'admin' in users %}...{% endif %}

B. {% if users contains 'admin' %}...{% endif %}

C. {% if 'admin' exists in users %}...{% endif %}

D. {% if users includes 'admin' %}...{% endif %}

Ansible Facts

What Are Ansible Facts?

They are the host-specific system data and properties to which users connect.

A fact can be the IP address, BIOS (basic input/output system) information, a system's software information, and even hardware information.

Accessing the Ansible Facts

Ansible facts uses the **setup** module to gather facts every time before running the playbooks.

Access Ansible facts using ad-hoc commands

The **setup** module in Ansible fetches system details from remote hosts to the controller node and displays them directly, making facts visible to users on the screen.

Syntax

```
ansible all -m setup
```

Filter out a specific value from Ansible facts

The **setup** module in Ansible fetches system facts, allowing users to display specific values using the filter argument.

Syntax

```
ansible all -m setup -a
"filter=ansible_cmdline"
```

Accessing the Ansible Facts

Access facts using Ansible playbook

Fetch the Ansible facts and display them using a playbook

Syntax

```
- hosts: all
      tasks:
      - debug:
          var: ansible_facts
```

Access a specific fact using an Ansible playbook

Fetch the Ansible facts, filter them, and display them using a playbook

Syntax

```
- hosts: all
      tasks:
      - debug:
          var:
ansible_facts["cmdline"]
```

Assisted Practice

Running Playbook Containing Ansible Facts **Duration: 15 Min.**

Problem statement:
You have been assigned the task of running a playbook that includes gathering facts (system information) and encounters issues such as incomplete or inaccurate data retrieval, leading to inconsistencies in subsequent automation tasks or deployments.

Outcome:
You can successfully gather accurate and comprehensive system facts during playbook execution, ensuring reliable information for efficient decision-making and automation processes, thereby improving overall operational efficiency.

Note: Refer to the demo document for detailed steps:
04_Running_Playbook_Containing_Ansible_Facts

Assisted Practice: Guidelines

Steps to be followed:

1. Run playbook containing facts

Variable File in Jinja2

Variable File

It is used to store and manage variables separately from the main templates.

Below are the key points:

Separation of concerns

Store variables separately to allow templates to focus on layout and design

Simplification of updates

Reuse variables across templates for easy and centralized changes

Variable File: Path

The **–vars-file** option provides the path to the file containing variables.

The file path must be one of the following:

01 Absolute file path

02 Relative to the project path

03 Relative to the Ansible folder

It is impossible to set variables inside a block and have them show up outside of it. This also applies to loops.

Variable File: Creation

It creates a separate YAML or JSON file to define variables used in Jinja2 templates.

Syntax

```
# filename.yml
variable_name: value
```

Example

```
# vars.yml
site_name: "My Webite"
admin_email: "admin@example.com"
```

Variable File: Example

The variable file incorporates predefined variables to dynamically render content, enhancing flexibility and reusability.

01 Variables.yml

```
site_name: "My Awesome Site"
admin_email: "admin@example.com"
```

02 Template.html

```
<h1>{{ site_name }}</h1>
<p>Contact us at: {{ admin_email }}</p>
```

Variable File: Example

03

Python

```
import yaml
from jinja2 import Environment, FileSystemLoader

# Load variables from the YAML file
with open('variables.yml', 'r') as file:
    variables = yaml.safe_load(file)

# Set up the Jinja2 environment
env =
Environment(loader=FileSystemLoader('templates'))
template = env.get_template('template.html')

# Render the template with the loaded variables
rendered_template = template.render(variables)

print(rendered_template)
```

This setup will dynamically insert the values from **Variables.yml** into **Template.html** when rendered by the Python script.

Quick Check

In your Ansible playbook, you need to include a file containing deployment-specific variables. The file is named deployment_vars.yml and is in the config's directory relative to your project path. How would you specify this file in your playbook?

A. -vars-file: "/absolute/configs/deployment_vars.yml"

B. -vars-file: "configs/deployment_vars.yml"

C. -vars-file: "../configs/deployment_vars.yml"

D. -vars-file: "~/configs/deployment_vars.yml"

Control Structure: Overview

Control Structure

It is used to manage the flow of template rendering and enable dynamic content insertion through loops and conditional statements.

01 Using loops

02 Using conditional control

Jinja2 syntax encloses the control structures inside the {% %} blocks.

Control Structure: Loops

01 Using loops

Jinja2 uses the **for statement** to provide looping functionality.

playbook

```
---
- hosts: worker1
  become: yes
  vars:
    usernames: ['Alice', 'Bob', 'John',
'Martin']

  tasks:
    - name: Loops usage within Jinja2
templates
      template:
        src: usernames.j2
        dest:
/home/ansible_user/usernames.txt
```

usernames.j2

```
The list of users who are going to
be part of the ongoing migration
process.

{% for item in usernames %}

   {{ item }}

{% endfor %}
```

Control Structure: Conditional Control

02 Using conditional control

Jinja2 uses the **if** statements for conditional control.

Syntax

```
{% if condition %}
    print_some_thing

{% elif condition2 %}
    execute_another_thing

{% else %}
    do_something_else

    {% endif %}
```

Assisted Practice

Combining Loops and Conditionals in Jinja2 **Duration: 15 Min.**

Problem statement:
You have been assigned the task of running a playbook that uses Jinja2 templates with loops and conditionals to generate dynamic configurations. However, you encounter issues, such as incorrect logic or syntax errors, leading to failed deployments and configuration inconsistencies.

Outcome:
You can successfully use loops and conditionals in Jinja2 templates during playbook execution, ensuring dynamic and accurate configurations for efficient decision-making and automation processes, thereby improving overall operational efficiency.

Note: Refer to the demo document for detailed steps:
05_Combining_Loops_and_Conditionals_in_Jinja2

Assisted Practice: Guidelines

Steps to be followed:

1. Prepare data and template files
2. Render and execute a template with a Python script

Quick Check

You are tasked with generating a report that includes a summary of sales data for each region. You need to iterate over a list of regions and display the sales figures for each. Which Jinja2 feature should you use to achieve this?

A. Blocks

B. Conditions

C. Loops

D. Macros

Tests in Jinja2

What Is Test in Jinja2?

It is a way of evaluating template expressions and returning **True** or **False**.

Difference between Tests and Filters:

Tests

Jinja Tests are used for comparisons.

Filters

Filters are used for data manipulation and have different applications in Jinja2.

Variable Testing

To test a variable or expression, add **is** followed by the name of the test after the variable.

Built-in Ansible tests return values such as **failed**, **changed**, **succeeded**, and **skipped.**

Input	Output
`variable is test_var`	`result is failed`

Tests can also accept arguments.

Tests: Example

Below is an example of matching strings against a substring or a regular expression, using the **match**, **search,** and **regex** test:

Example

```
vars:
  age: 25

tasks:
  - debug:
      msg: "User is an adult"
    when: age >= 18

  - debug:
      msg: "User is a teenager"
    when: age >= 13 and age < 18

  - debug:
      msg: "User is a child"
    when: age < 13
```

Tests: Comparing Versions

The following is an example of comparing versions:

Example

```
vars:
    my_version: 1.2.3
tasks:
    - debug:
        msg: "my_version is higher than 1.0.0"
      when: my_version is version('1.0.0', '>')
```

If **my_version** is greater than or equal to **1.0.0**, this test returns True; otherwise, it is False.

Note

When using a **version** in a playbook or role, do not use {{ }}.

Ansible Built-in Tests

Below are some of the Ansible built-in tests:

boolean()	even()	in()	mapping()	sequence()
callable()	false()	integer()	ne()	string()
defined()	filter()	iterable()	none()	test()
divisibleby()	float()	le()	number()	true()
eq()	ge()	lower()	odd()	undefined()
escaped()	gt()	lt()	sameas()	upper()

Quick Check

You need to run a debug task only if the variable my_version is 1.2.3 or higher. How would you write the condition in your Ansible playbook?

A. my_version == '1.2.3'

B. my_version >= '1.2.3'

C. my_version is version('1.2.3', '>=')

D. my_version >= version('1.2.3')

Key Takeaways

- The Jinja2 template helps in creating configuration files for servers that require different configurations.
- A variable in Jinja2 is a placeholder used to store and manage dynamic data within a template.
- Filters are used to alter the appearance of the output or format data by piping the variable name.
- Array is a data structure that store multiple values in a single variable, represented as lists.
- Ansible facts uses the **setup** module for gathering facts every time before running the playbooks.

Key Takeaways

- In variable files, the –vars-file option provides the path to the file containing variables.
- Jinja2 templating in Ansible streamlines playbook creation through loops and conditional statements for dynamic content insertion.
- Test in Jinja2 is a way of evaluating template expressions and returning **True** or **False**.

Templating with Jinja2

Duration: 25 Min.

Project agenda: To demonstrate the use of Jinja2 templates in Ansible for various tasks, including variable substitution, data manipulation, applying filters, and iterating over lists, by embedding variables within double curly braces

Description: As a systems administrator, you must create Ansible playbooks to demonstrate Jinja2 template features. This project will cover defining variables, data manipulation, using default filters, iterating over lists, and applying filters for pathname, date, and time. You'll work on real-world scenarios to master Jinja2 templates in Ansible for effective configuration management automation.

LESSON-END PROJECT

Templating with Jinja2

Duration: 25 Min.

Perform the following:

1. Create a new directory
2. Define a variable inside a playbook using vars
3. Perform data manipulation using filters
4. Use default filters
5. Iterate over the individual values of the list and perform the operation
6. Use filters when dealing with pathnames
7. Use filters for date and time

Expected Deliverables: Ansible playbooks demonstrating Jinja2 template features, including variable substitution, data manipulation, applying filters, and iterating over lists. Documentation detailing the steps and outcomes of each playbook.

LESSON-END PROJECT

Thank You