# Ansible Playbooks

Ansible Fundamentals

# Agenda

- YAML overview
- General Playbook Structure
- Idempotent Playbooks
- Commonly Used Modules
- Task Results
- Validating the Result

# YAML overview

Ansible Playbooks

# YAML Overview: Basics

- Format
  - YAML stands for "Yet Another Markup Language."
  - It's a human-readable data serialization format
- Indentation
  - Uses spaces (not tabs) for indentation, which denotes hierarchy
  - Most of the issues with YAML is about indentation
- Case Sensitive
  - YAML is case sensitive
- YAML File is a collection of key-value pairs

# YAML Overview: Data Types

- Scalars
  - Single values, which can be strings, numbers, or booleans
- Mappings
  - Key-value pairs, similar to dictionaries or hashes in other languages
  - Denoted with **`key: value`** format
- Lists
  - Ordered sequences of values
  - Each item in a list is denoted with a - (dash) followed by a space

# YAML Overview: Document Start/End

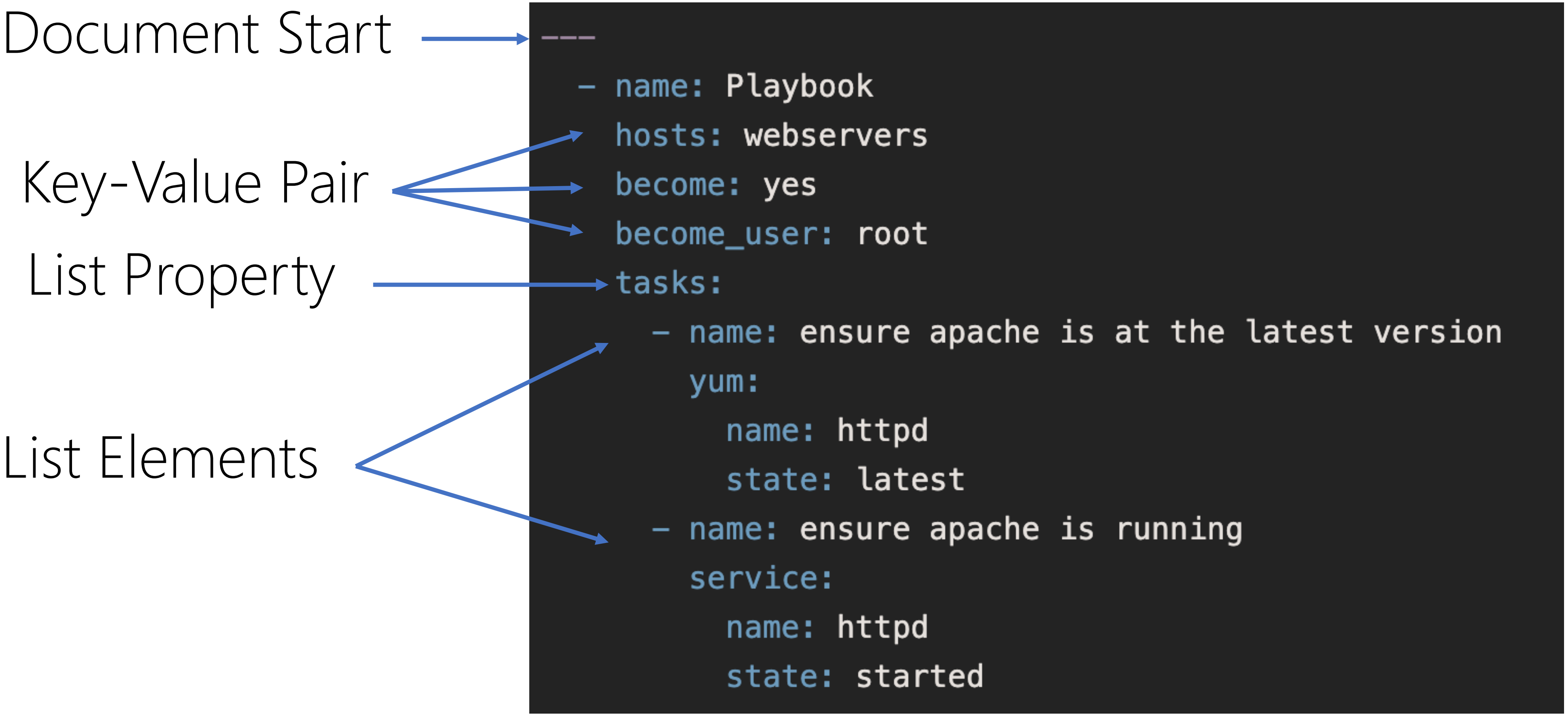
- Start
  - An optional `---` at the beginning indicates the start of a YAML document
- End:
  - An optional `...` at the end indicates the end of a YAML document
- If you want to add more than one playbook on a file, you need to use the start element (`---`) to separate the objects

# YAML Overview: Strings and comments

- Quotation
  - Strings can be written with or without quotes
  - However, for strings containing special characters or reserved words, it's safer to use single or double quotes
- Multiline
  - Use the **>** character for folded style (newlines become spaces)
  - Use **|** for literal style (newlines are preserved)
- Comments
  - Use **#** to add comments
  - Everything after **#** on that line is a comment.

- More here: https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html#yaml-syntax

# YAML Overview: Document Start/End



Document Start

Key-Value Pair

List Property

List Elements

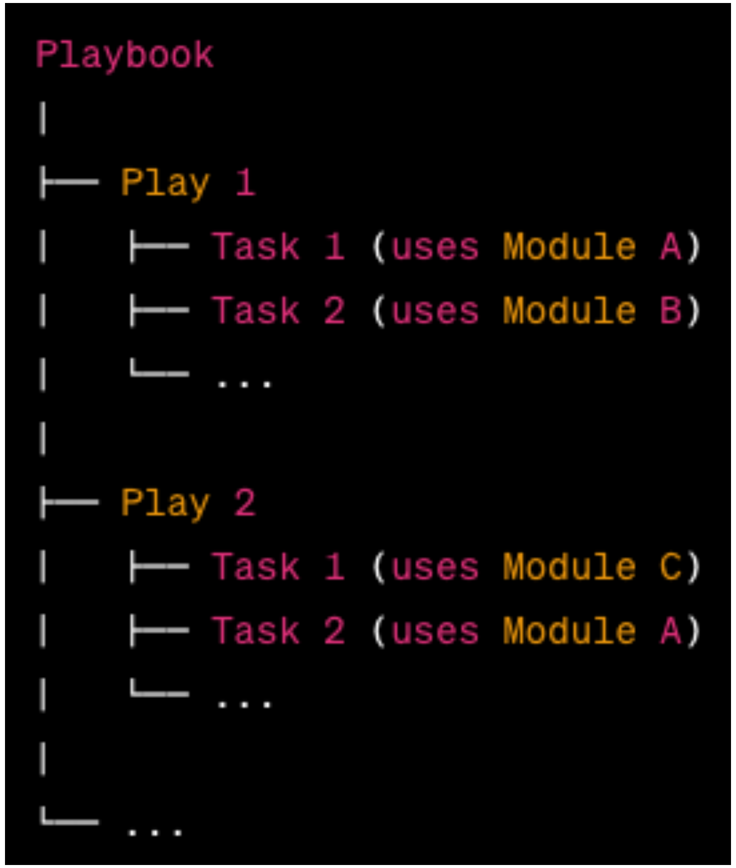
```yaml
---
  - name: Playbook
    hosts: webservers
    become: yes
    become_user: root
    tasks:
      - name: ensure apache is at the latest version
        yum:
          name: httpd
          state: latest
      - name: ensure apache is running
        service:
          name: httpd
          state: started
```

# General Playbook Structure

Ansible Playbooks

# Modules, Tasks, Plays, Playbooks

- A **playbook** orchestrates multiple plays, defining the broader automation workflow.
- A **play** is a collection of tasks executed on a group of hosts.
- A **task** uses a module to execute that action with specific parameters.
- A **module** is a tool that performs a specific action.



```
Playbook
│
├── Play 1
│   ├── Task 1 (uses Module A)
│   ├── Task 2 (uses Module B)
│   └── ...
│
├── Play 2
│   ├── Task 1 (uses Module C)
│   ├── Task 2 (uses Module A)
│   └── ...
│
└── ...
```

# Playbook

- A playbook is a YAML file that contains one or more plays
- It provides a script-like experience, where multiple plays are executed in order, each with its set of tasks.
- Relationship
  - The playbook is the top-level component
    - It orchestrates the execution of plays, which in turn run tasks that call upon modules

# Play

- A play is a set of tasks that will be run on a particular set of hosts in a sequence
- It defines which hosts from the inventory the tasks should run on and sets variables that can be used in the tasks.
- Relationship
  - Plays organize tasks
  - A single playbook can contain multiple plays, allowing for different sets of tasks to be run on different hosts or groups of hosts.

# Tasks

- Tasks define a single action that will be executed on the target host
- Each task calls an Ansible module with specific arguments
- Relationship
    - A task is essentially an instance of a module with specific parameters
    - Multiple tasks together form the actions in a play.

# Modules

- Modules are the units of work in Ansible
- They are like command-line tools but can be run directly or through a playbook
- Each module is designed to accomplish a specific task, such as managing packages, creating users, or interacting with APIs
- Relationship
  - Modules are the building blocks that tasks use to perform actions.

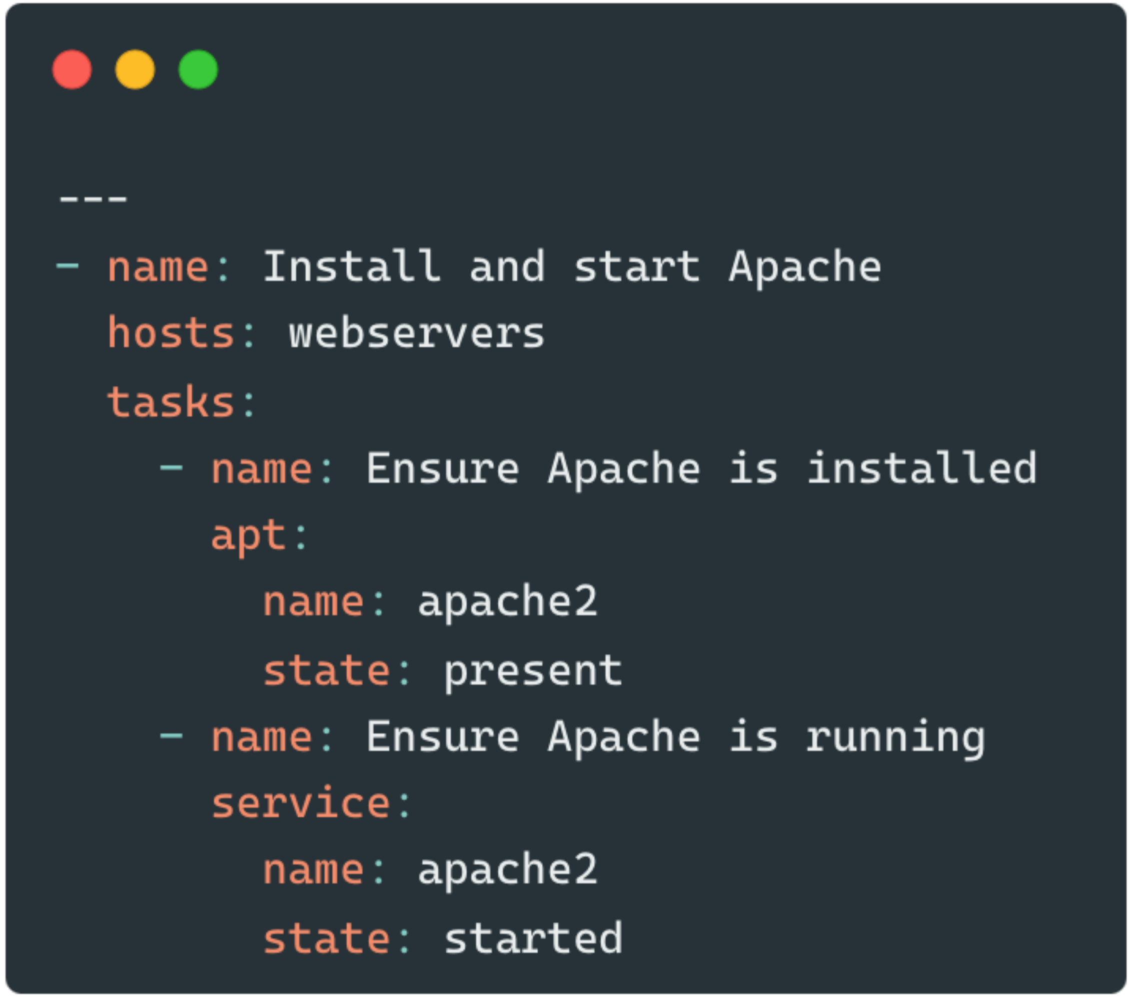
# Variables

- Modules are the units of work in Ansible
- They are like command-line tools but can be run directly or through a playbook
- Each module is designed to accomplish a specific task, such as managing packages, creating users, or interacting with APIs
- Relationship
    - Modules are the building blocks that tasks use to perform actions.

# Basic Play Structure

- **`name`**
  - Specify play name
  - Important for identify on logs
- **`hosts`**
  - Specifies which hosts the tasks will run on
  - Can target individual hosts, groups, or patterns
- **`tasks`**
  - A list of tasks to execute in order
  - Each task calls an Ansible module.

# Basic Playbook



```yaml
---
- name: Install and start Apache
  hosts: webservers
  tasks:
    - name: Ensure Apache is installed
      apt:
        name: apache2
        state: present
    - name: Ensure Apache is running
      service:
        name: apache2
        state: started
```

# Sequential Execution

- Plays
  - In a playbook, plays are executed sequentially
  - If you have multiple plays in a playbook, the first play will run to completion on all targeted hosts before the second play starts, and so on
- Tasks
  - Within a play, tasks are also executed sequentially
  - The first task will run on all targeted hosts before the second task starts, and so on
  - Inside one task, several hosts run on parallel

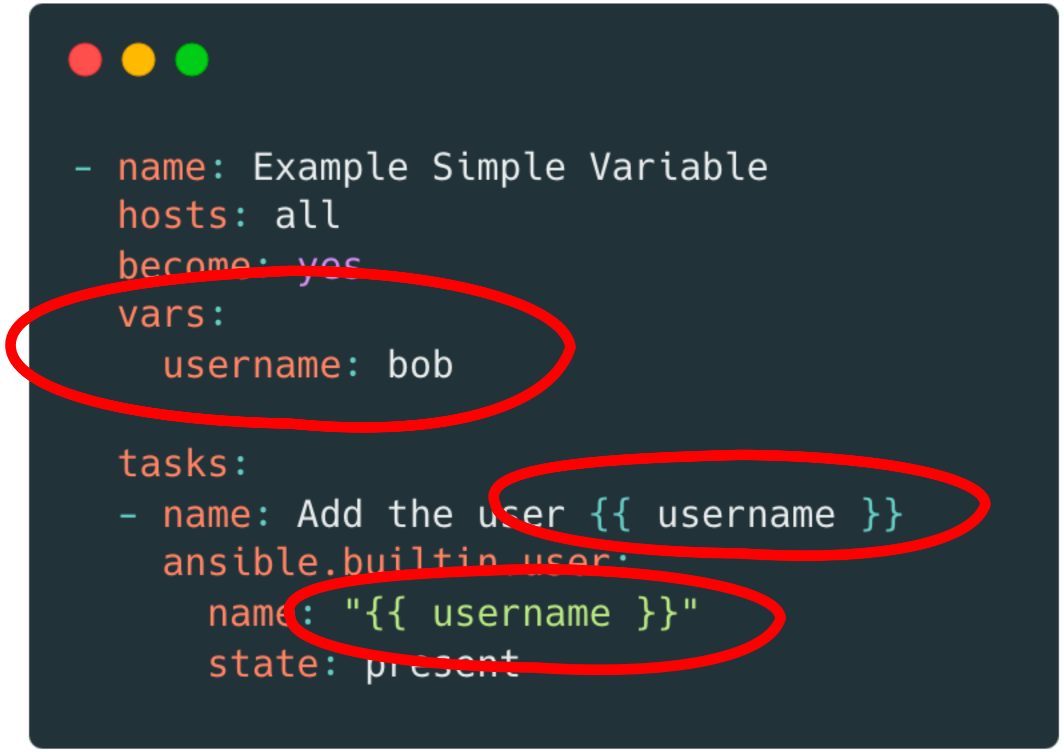
# Variables

- You can define them directly on the playbook, using group variables or host variables
- To reference a variable you may user the format **`{{ var_name }}`**
- Ansible already have some built-in variables that can grant you some context variables
  - **`inventory_hostname`**
  - **`hostvars`**
  - **`ansible_play_name`**
- Complete list: [https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html](https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html)

# Tasks Structure

- **`name:`**
  - A human-readable description of the task
- Module
  - The action to be taken, using an Ansible module
  - This parameter uses the module name directly
- **`args/vars`**
  - Arguments or parameters for the module

# Playbook with Variables

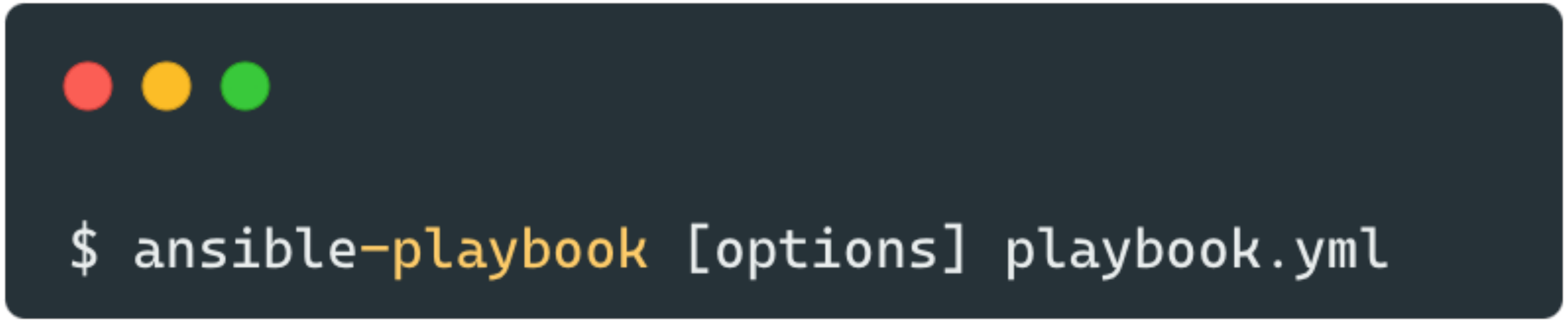


```
- name: Example Simple Variable
  hosts: all
  become: yes
  vars:
    username: bob

  tasks:
  - name: Add the user {{ username }}
    ansible.builtin.user:
      name: "{{ username }}"
      state: present
```

# Execute Playbook

- Using ansible-playbook command



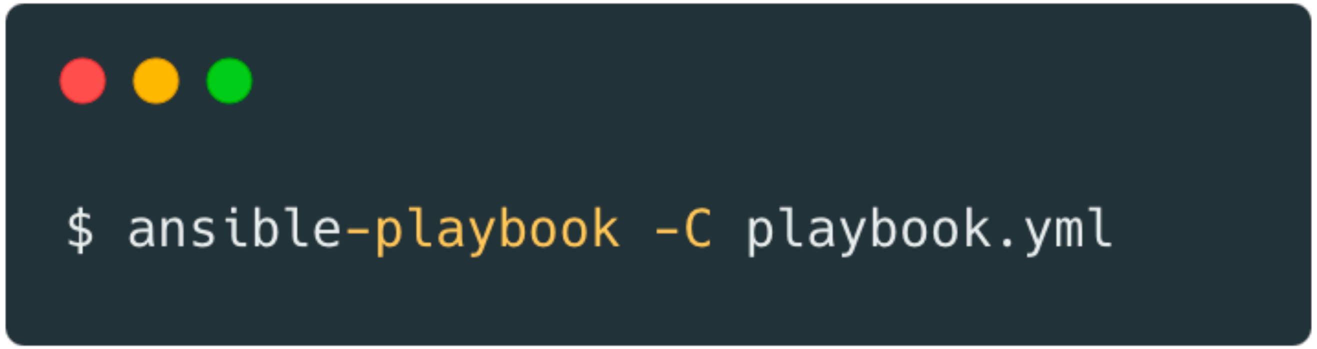
```
$ ansible-playbook [options] playbook.yml
```

# Common Options

- Same as ad-hoc commands
- `-i` or `--inventory`: Specify the location of the inventory file
- `-u` or `--user`: Define the remote user to execute tasks as. By default, it uses the current user
- `-k`: Ask for SSH password instead of using key-based authentication
- `-b` or `--become`: Allows privilege escalation (e.g., using sudo). Useful if tasks need root privileges
- `--ask-become-pass` or `-K`: Ask for privilege escalation password (e.g., sudo password)
- `-v` to `-vvvv`: Increase verbosity. More "v"s give more detailed output
- `--check` or `-C`: Run in check mode. Ansible will not make any changes on the hosts, but will simulate the execution to show what would have changed

# Execute Playbook with DryRun

- Using ansible-playbook command with –C flag allow you to run on dry-run mode
- This mode don’t so any change but can list you all possible changes if you really execute the playbook



```
$ ansible-playbook -C playbook.yml
```

# Run your First Playbook

Demo

# Idempotent Playbooks

Ansible Playbooks

# Understanding Idempotency

- Writing idempotent tasks is a fundamental principle in Ansible
- Ensures that running your playbook multiple times doesn't change the system state after the first run, unless the system state has changed in the meantime
- A task is idempotent if it can be applied multiple times without changing the result beyond the initial application
- Ensures consistency, avoids unintended side-effects, and makes playbooks safe to run repeatedly

# Imperative Configuration

- In an imperative approach, you specify how to achieve a particular state, detailing each step
- Concentrates on the process and sequence of operations to achieve the desired result
- Offers more control and can be more flexible in certain scenarios, as you dictate the exact sequence of operations.
- Example: Traditional shell scripts or batch scripts where you list each command to run in sequence are imperative.

# Declarative Configuration

- In a declarative approach, you specify what you want the system to look like, not how to achieve that state
- Concentrates on the desired end state
- The system or tool figures out the necessary steps to reach that state
- Often simpler and more readable, as you don't need to specify every step
- Reduces the chance of errors since the tool handles the process.
- Example: Ansible playbooks, Terraform configurations, and Kubernetes manifests are primarily declarative

# Declarative vs Imperative

- Clarity vs. Control
  - Declarative configurations are often clearer and more concise, focusing on the "what”
  - Imperative configurations give more control by focusing on the "how"
- Tool Responsibility
  - In declarative configurations, the tool is responsible for figuring out how to achieve the desired state, reducing potential errors
  - In imperative configurations, the responsibility lies more with the developer or operator
- Flexibility
  - While declarative tools are designed for specific use cases (e.g., Ansible for configuration management), imperative approaches can be more flexible and can handle a wider range of tasks
- Learning Curve
  - Declarative tools might have a steeper initial learning curve as users need to understand the tool's conventions and capabilities
  - Imperative approaches, being more manual, might be more intuitive initially but can become complex as tasks grow

# Use Ansible Modules Properly

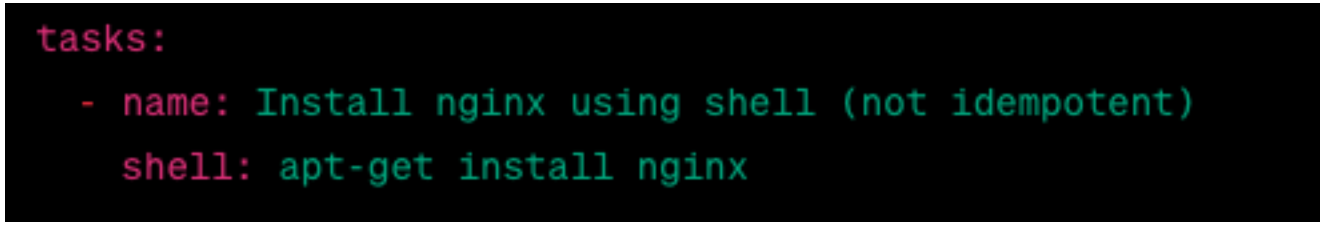
- Commands like `shell` or `command` are not inherently idempotent
- If you must use them, ensure idempotency by adding conditions
- Most Ansible modules are designed to be idempotent
- Always prefer using a module over running raw commands
- For example, use the file module to manage files instead of raw shell or command tasks.

# Test with Check Mode

- Run playbooks with **`--check`** (check mode) to see what changes would be made without actually applying them
- A truly idempotent task will not report changes on subsequent runs unless the system state has changed

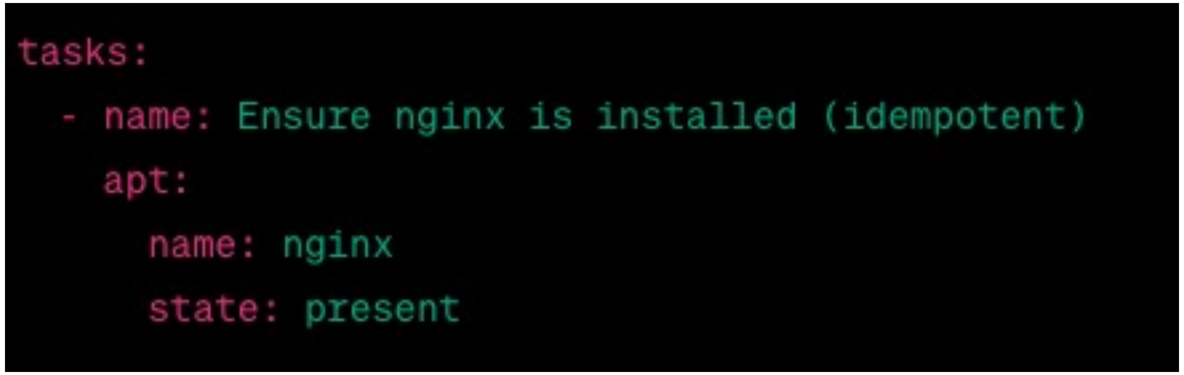
# Non-idempotent vs Idenpontent way

- Non-idempotent way



```
tasks:
  - name: Install nginx using shell (not idempotent)
    shell: apt-get install nginx
```

- Idempotent way



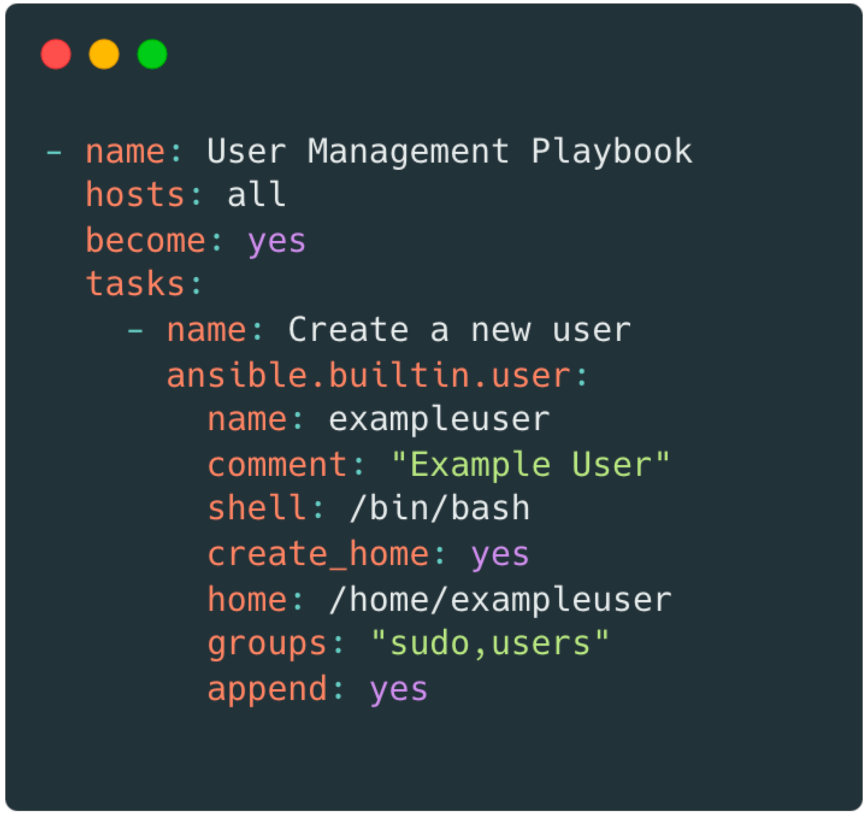
```
tasks:
  - name: Ensure nginx is installed (idempotent)
    apt:
      name: nginx
      state: present
```

# Commonly used Modules

Ansible Playbooks

# User Module

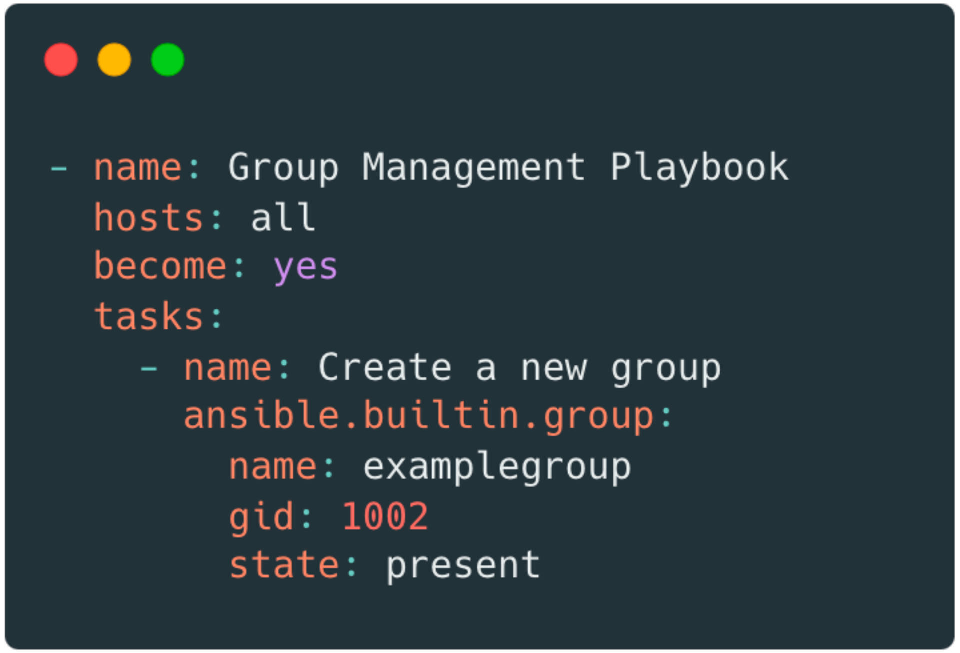
- Manage user accounts:
- https://docs.ansible.com/ansible/latest/collections/ansible/builtin/user_module.html



```yaml
- name: User Management Playbook
  hosts: all
  become: yes
  tasks:
    - name: Create a new user
      ansible.builtin.user:
        name: exampleuser
        comment: "Example User"
        shell: /bin/bash
        create_home: yes
        home: /home/exampleuser
        groups: "sudo,users"
        append: yes
```

# Group Module

- Manage group accounts:
- https://docs.ansible.com/ansible/latest/collections/ansible/builtin/group_module.html

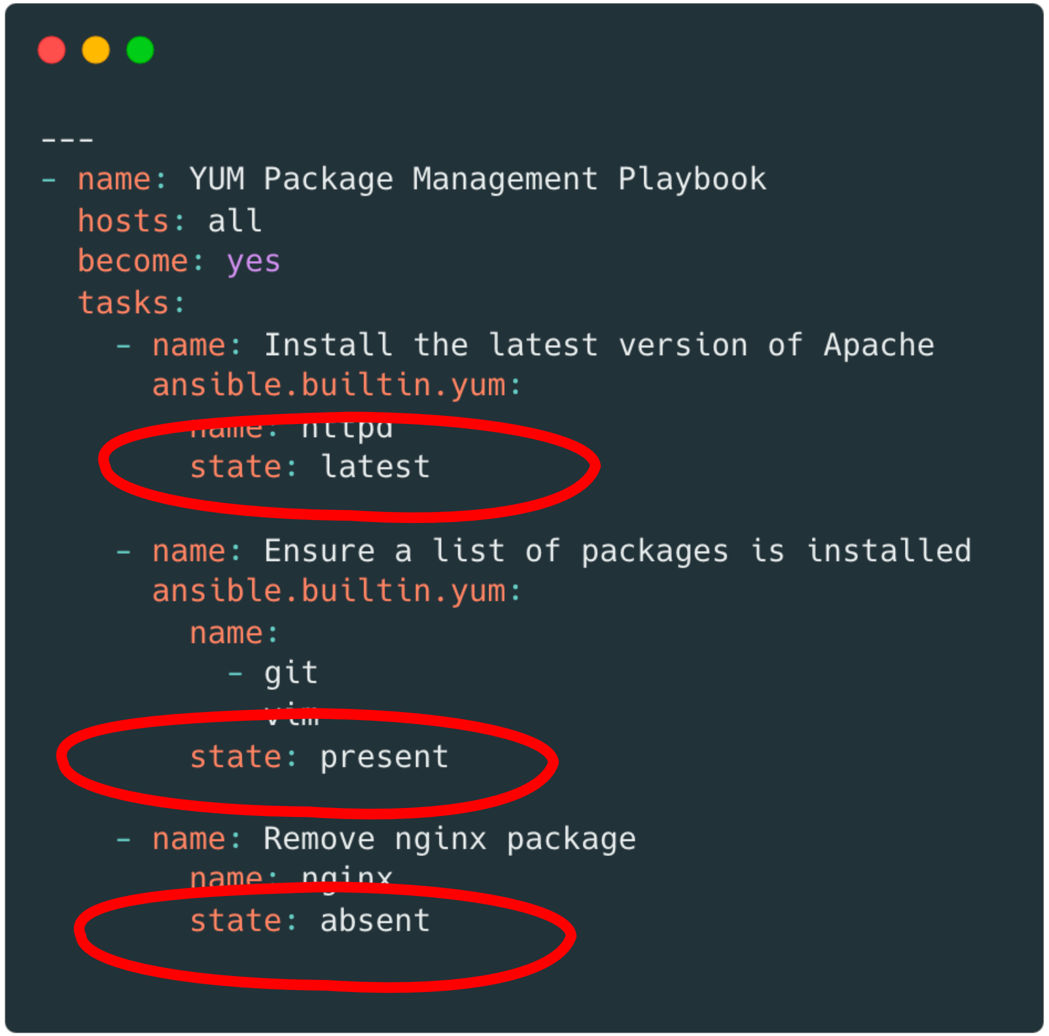


```
- name: Group Management Playbook
  hosts: all
  become: yes
  tasks:
    - name: Create a new group
      ansible.builtin.group:
        name: examplegroup
        gid: 1002
        state: present
```

# Yum Module (Package management)

- Manage yum packages: ansible.builtin.yum module – Manages packages with the yum package manager — Ansible Documentation

- You can find modules for several package managers



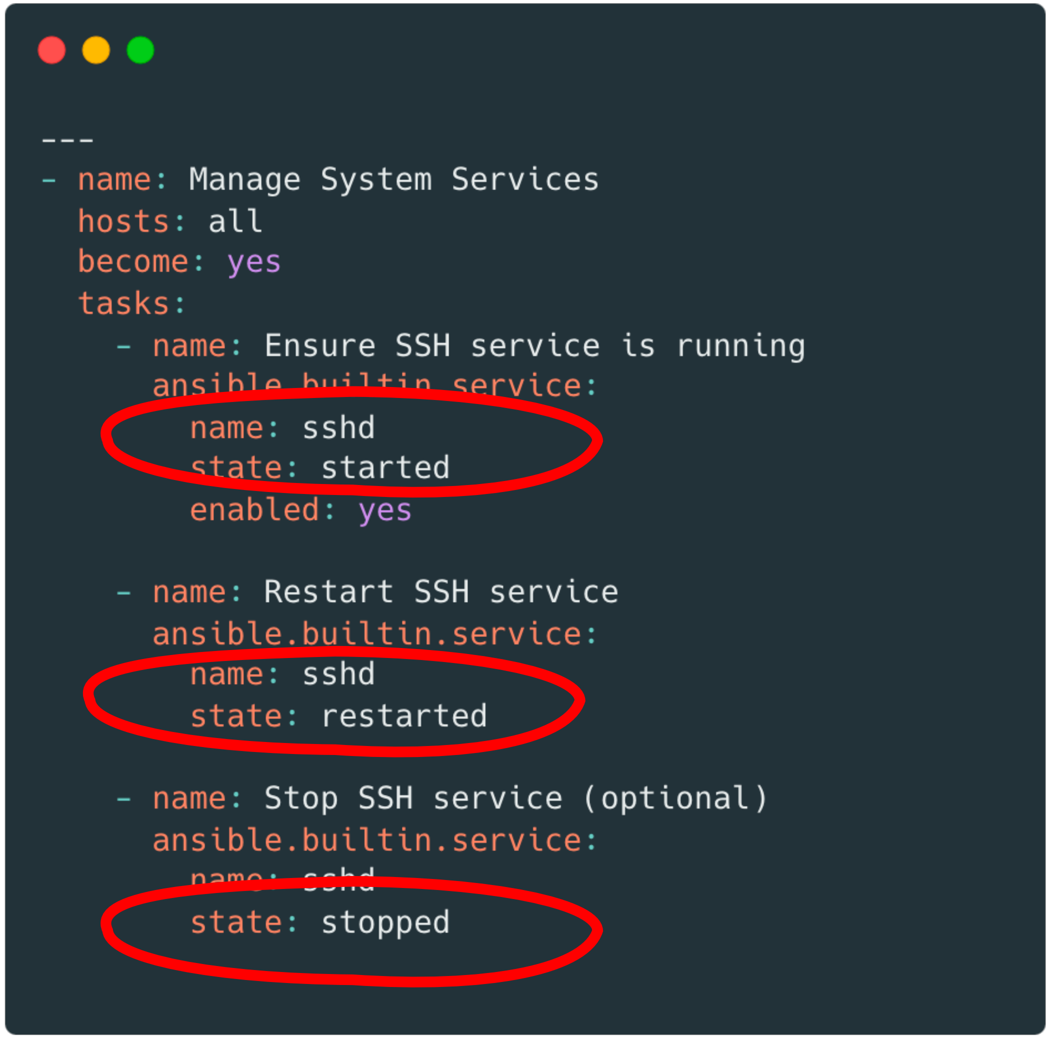
```yaml
---
- name: YUM Package Management Playbook
  hosts: all
  become: yes
  tasks:
    - name: Install the latest version of Apache
      ansible.builtin.yum:
        name: httpd
        state: latest

    - name: Ensure a list of packages is installed
      ansible.builtin.yum:
        name:
          - git
          - vim
        state: present

    - name: Remove nginx package
        name: nginx
        state: absent
```

# Service Module

- Manage services: https://docs.ansible.com/ansible/latest/collections/ansible/builtin/service_module.html



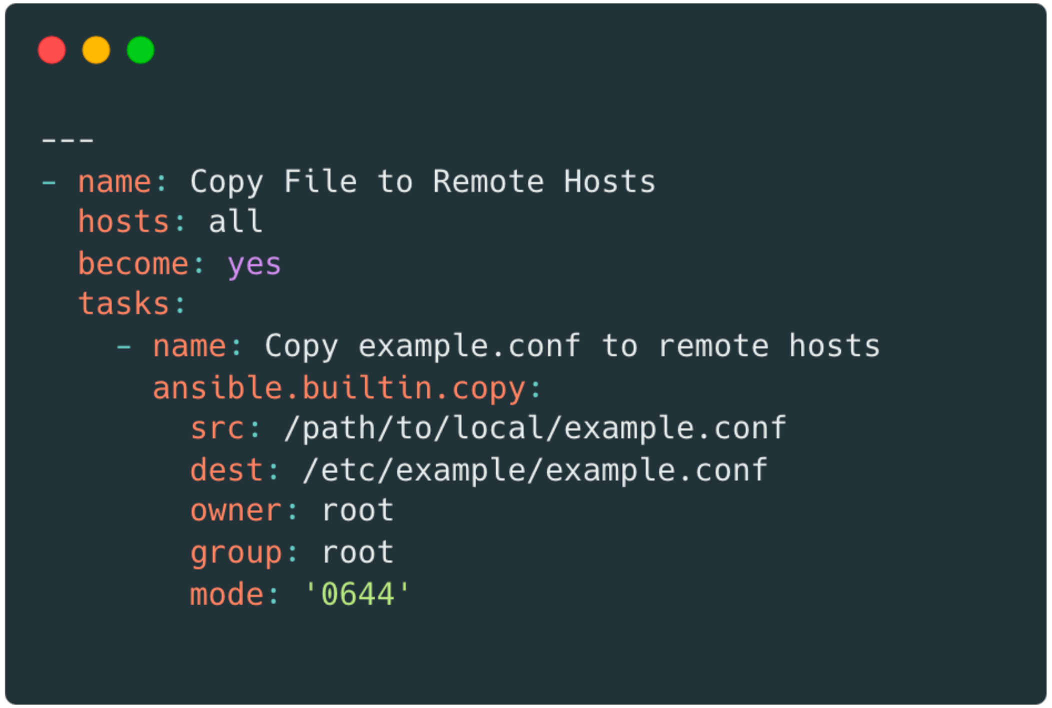
```yaml
---
- name: Manage System Services
  hosts: all
  become: yes
  tasks:
    - name: Ensure SSH service is running
      ansible.builtin.service:
        name: sshd
        state: started
        enabled: yes

    - name: Restart SSH service
      ansible.builtin.service:
        name: sshd
        state: restarted

    - name: Stop SSH service (optional)
      ansible.builtin.service:
        name: sshd
        state: stopped
```

# Copy Module

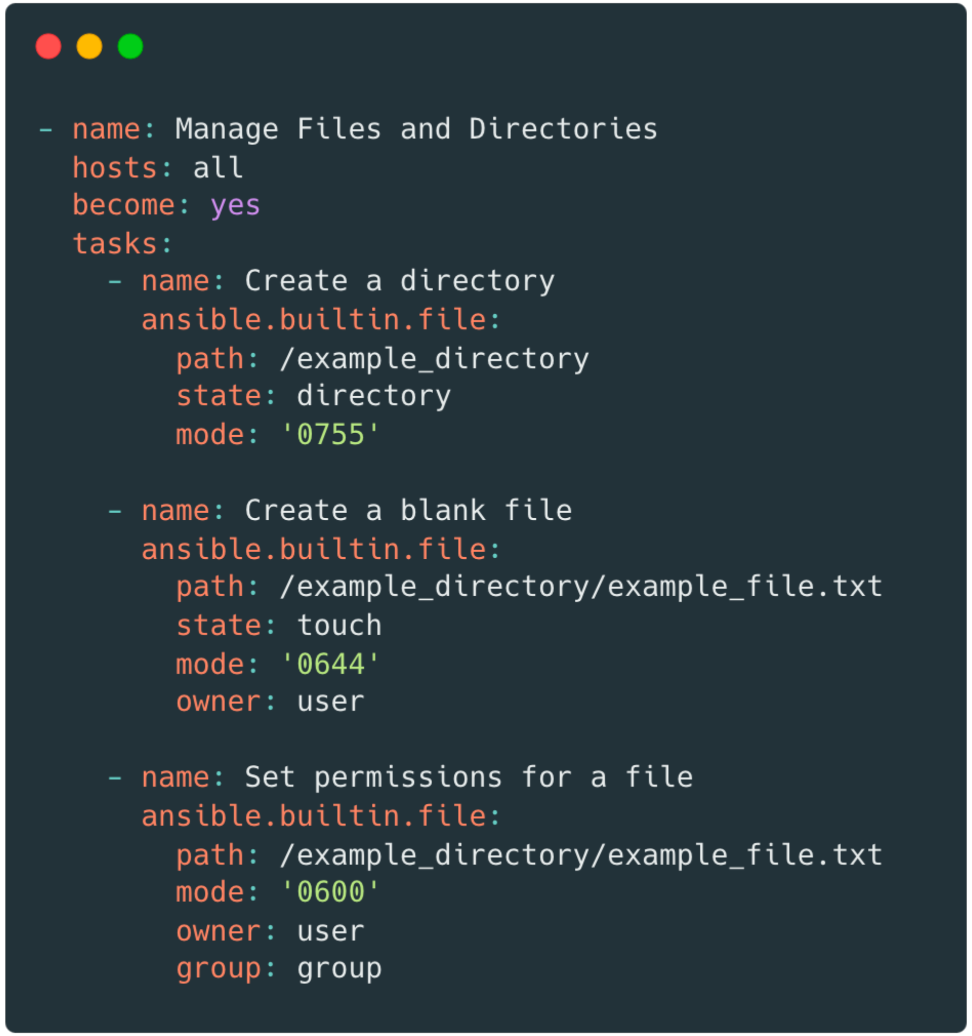
- Copies files from the local to a location on the remote machine
- https://docs.ansible.com/ansible/latest/collections/ansible/builtin/copy_module.html#ansible-collections-ansible-builtin-copy-module



```yaml
---
- name: Copy File to Remote Hosts
  hosts: all
  become: yes
  tasks:
    - name: Copy example.conf to remote hosts
      ansible.builtin.copy:
        src: /path/to/local/example.conf
        dest: /etc/example/example.conf
        owner: root
        group: root
        mode: '0644'
```

# File Module

- Manages file properties
- https://docs.ansible.com/ansible/latest/collections/ansible/builtin/file_module.html#ansible-collections-ansible-builtin-file-module



```yaml
- name: Manage Files and Directories
  hosts: all
  become: yes
  tasks:
    - name: Create a directory
      ansible.builtin.file:
        path: /example_directory
        state: directory
        mode: '0755'

    - name: Create a blank file
      ansible.builtin.file:
        path: /example_directory/example_file.txt
        state: touch
        mode: '0644'
        owner: user

    - name: Set permissions for a file
      ansible.builtin.file:
        path: /example_directory/example_file.txt
        mode: '0600'
        owner: user
        group: group
```

# Task Results

Ansible Playbooks

# Task Results

- Every time you execute a task, you get a result
- Possible results
  - OK
  - Changed
  - Failed
  - Skipped
  - Unreachable

# Task Results: OK

- The task executed successfully
- The module ran without any errors, and the desired state expressed in the task is already in place on the target system
- In other words, the system was already in the desired state, so no changes were made.
- Example: If you have a task to ensure a package is installed, and the package is already installed, the task result will be "OK".

# Task Results: Changed

- The task executed successfully and made changes to the target system
- The module ran without any errors, and the system was not initially in the desired state, so Ansible made the necessary changes to bring the system to that state.
- Example: If you have a task to ensure a package is installed, and the package was not initially installed, Ansible will install it, and the task result will be "changed".

# Task Results: Failed

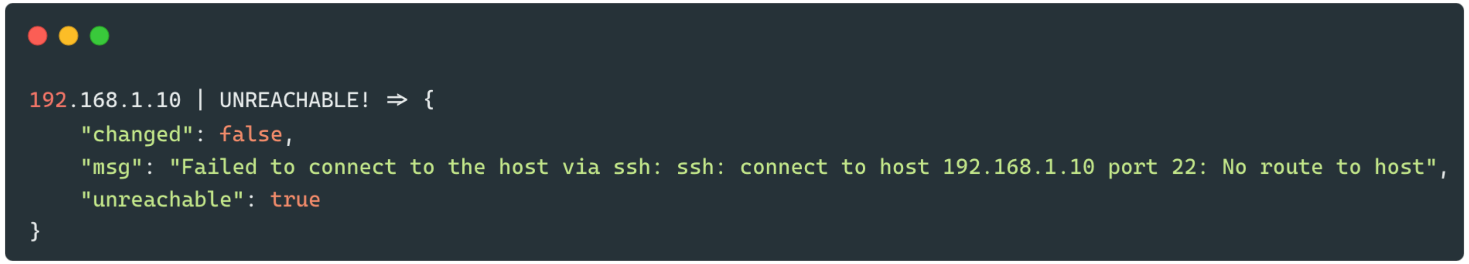
- The task did not execute successfully and encountered an error.
- An error occurred that prevented the module from completing its operation.
- This could be due to various reasons like incorrect parameters, issues on the target system, unreachable hosts, etc.
- Example: If you have a task to ensure a package is installed, but there's an issue with the package repository or network connectivity, the task might fail to install the package, resulting in a "failed" state.

# Task Results: Skipped

- The task was intentionally not executed on a particular host
- Tasks can be conditionally executed based on the evaluation of a when clause
- If the condition in the when clause evaluates to false, the task will be skipped for that host
- Example: If you run the following task on a RedHat system, the result will be ”Skipped”

# Task Results: Unreachable

- Ansible was unable to establish a connection to the target host
- This state typically indicates a fundamental communication issue between the Ansible control node and the target host
- Common reasons include network connectivity problems, incorrect SSH configurations, SSH key mismatches, host firewalls blocking access, or the target host being down
- When a host is in an "Unreachable" state, Ansible will not attempt any further tasks on that host for the duration of the playbook run



```
192.168.1.10 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: ssh: connect to host 192.168.1.10 port 22: No route to host",
    "unreachable": true
}
```

# Validating the Result

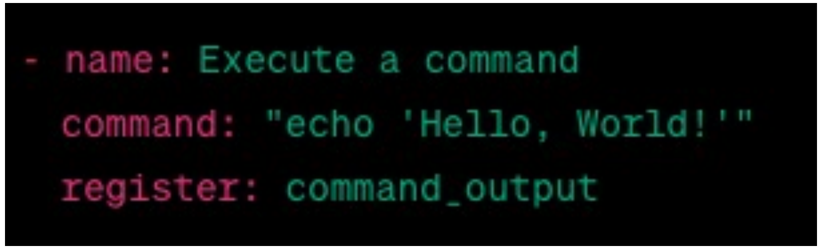
Ansible Playbooks

# Validating Results

- You may validate the results of a task and use that results on following tasks
- Usually, you start to save task output to a variable
- Then you may use variable content on other tasks to print values or decide about task execution

# Getting task output

- Use the **`register`** keyword to save the output of a task to a variable

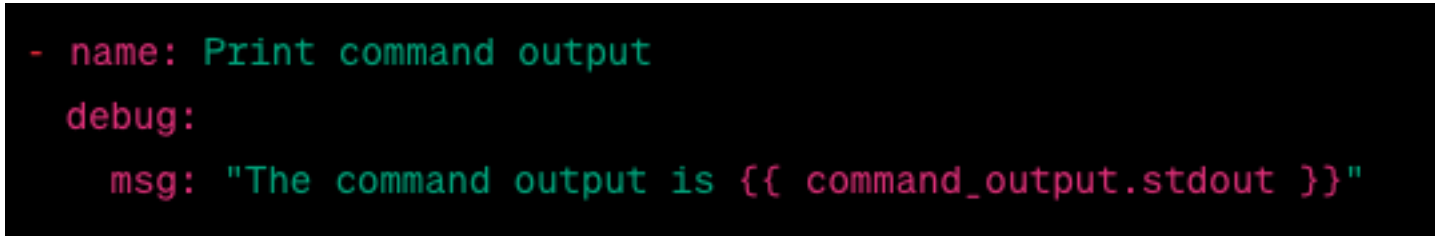


```
- name: Execute a command
  command: "echo 'Hello, World!'"
  register: command_output
```

- Then you can use variable attributes as a common variable
- Each task (module) will add specific attributes
- Common attributes include
  - **`command_output.stdout`**: The standard output of the command
  - **`command_output.stderr`**: The standard error of the command
  - **`command_output.rc`**: The return code of the command
  - **`command_output.changed`**: Boolean indicating if the task made changes

# Debugging outputs

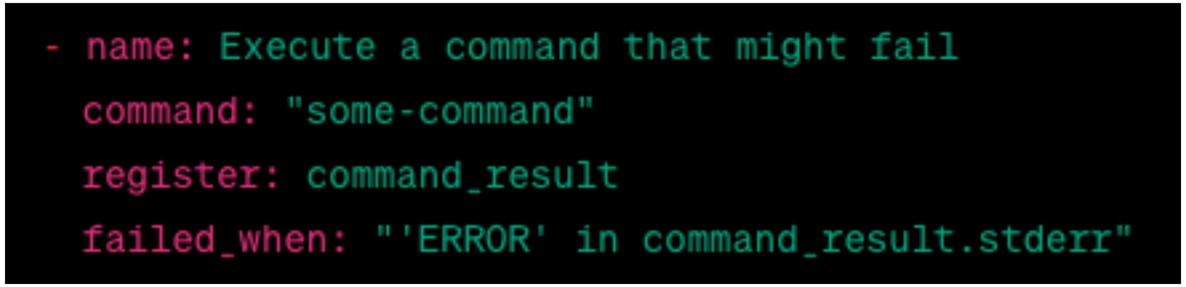
- Print messages, variables, or task results for debugging purposes



```
- name: Print command output
  debug:
    msg: "The command output is {{ command_output.stdout }}"
```

# Handling Failures Manually

- Customize when Ansible should consider a task as failed using the **`failed_when`** keyword.



```
- name: Execute a command that might fail
  command: "some-command"
  register: command_result
  failed_when: "'ERROR' in command_result.stderr"
```

# Use Task Results

Demo