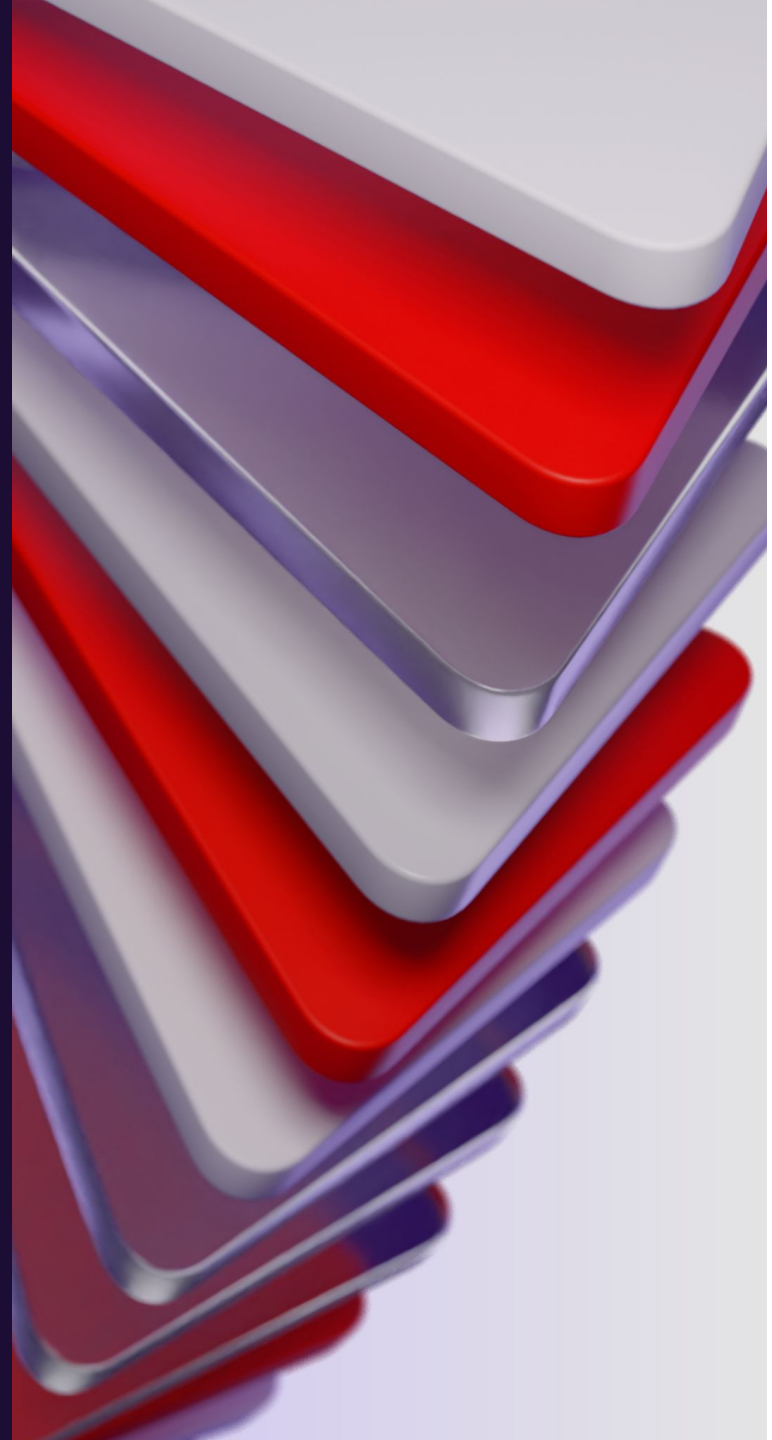


Red Hat
Ansible Automation
Platform

Red Hat Ansible Automation Platform

RH294

Francesco Marchioni
RHCA

Red Hat

Ansible Automation Platform

Introduction to Ansible Automation Platform

What is Ansible ?

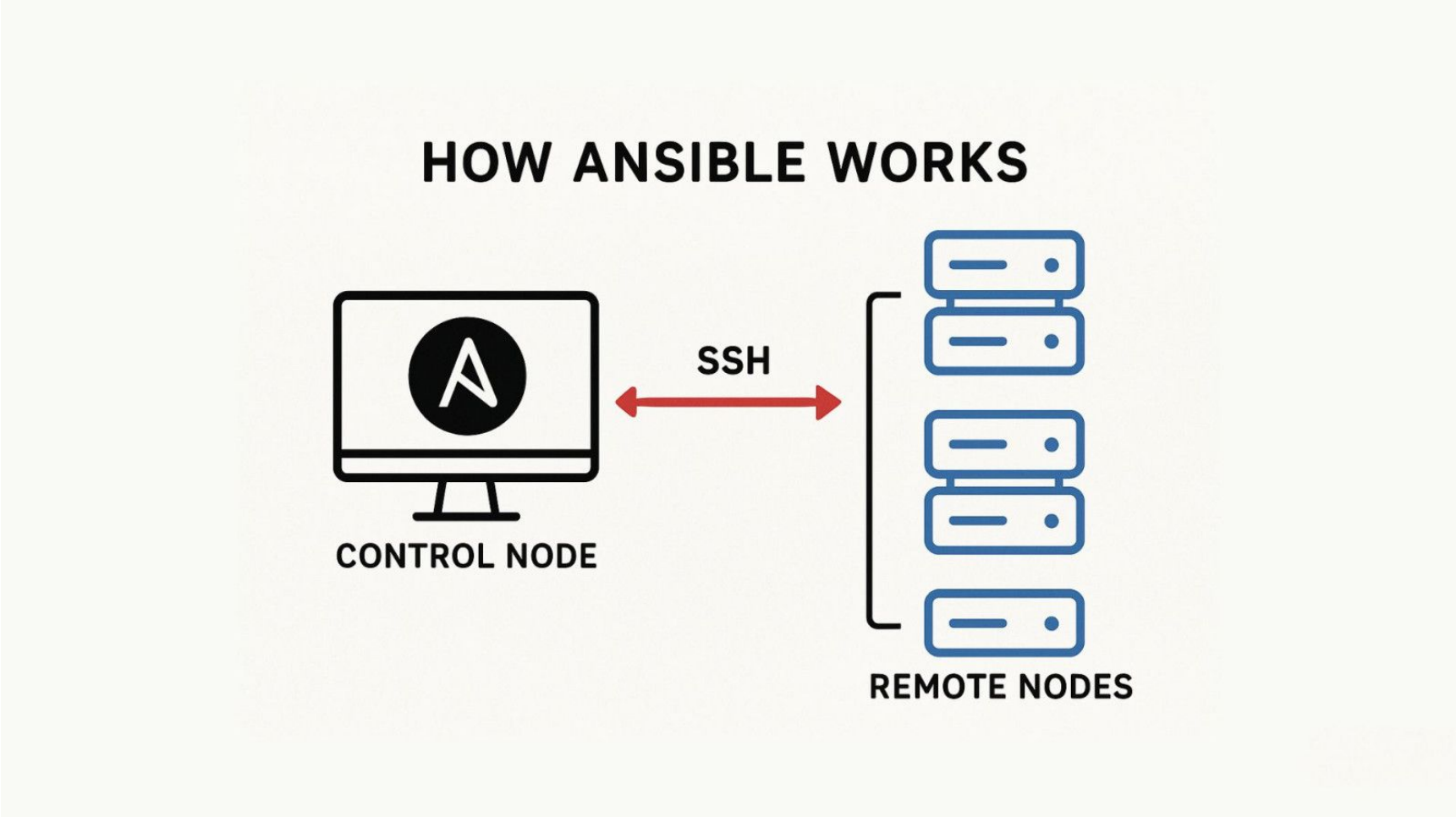
- Ansible is not about programming.
- Ansible is an Open Source Automation Platform
- You don't tell Ansible what to do.
- **You tell it what you want.....**
- **....even.....**

Ansible Automation Platform

We will steal the Moon !

```yaml
---
- name: Steal the Moon
  hosts: moon_base
  become: true
  vars:
    shrink_ray_path: /opt/weapons/shrink_ray
    containment_unit: /var/vault/moon.jar

  tasks:
    - name: Locate the moon
      ansible.builtin.debug:
        msg: "Target acquired: The Moon is in geostationary orbit."

    - name: Fire shrink ray
      ansible.builtin.command:
        cmd: "{{ shrink_ray_path }} --target=moon"
      register: ray_status

    - name: Confirm moon size reduction
      ansible.builtin.debug:
        msg: "Shrink ray status: {{ ray_status.stdout }}"

    - name: Store moon in containment unit
      ansible.builtin.copy:
        src: /tmp/shrunken_moon
        dest: "{{ containment_unit }}"

    - name: Mission accomplished
      ansible.builtin.debug:
        msg: "The Moon is now yours. Well done, Gru!"
```

Ansible Automation Platform



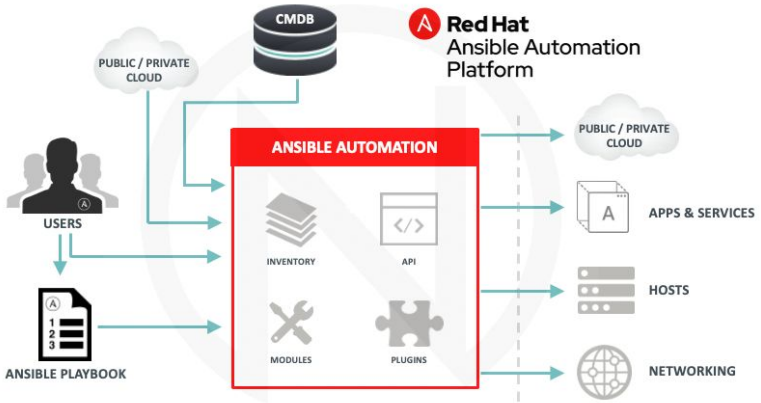
Ansible Automation Platform

Ansible Automation Platform

Key components

Ansible Automation Platform is composed of multiple key components designed to manage and scale automation in enterprise environments:

- Ansible Core
- Ansible Content Collections
- Automation Content Navigator
- Automation Execution Environments
- Automation Controller
- Automation Hub



Ansible Core

- Provides the foundation of Ansible automation.
- Defines the YAML-based automation language used in Playbooks.
- Includes core capabilities:
  - Loops, conditionals, variables
  - Task execution and control flow
- Includes CLI tools: ansible, ansible-playbook, etc.
- Distributed as the ansible-core RPM.

```
$ ansible  -- version
```

Ansible Content Collections

- Modular packaging of Ansible content:
  - Modules, Roles, and Plugins
- Replaces "batteries-included" monolithic module strategy.
- Enables:
  - Versioning and independent development
  - Simplified content reuse and contribution
- ansible.builtin is the only collection shipped in Ansible Core.
- Red Hat provides over 120+ Certified Collections.
- Additional community content is available via Ansible Galaxy.

Ansible Content Navigator (ansible-navigator)

- Unified CLI tool to interact with Ansible environments.
- Replaces tools like ansible-playbook, ansible-inventory, etc.
- Key features:
  - Interactive and text-based UI
  - Integrated with execution environments
- Supports local development and testing
- Enables separation between:
  - Developer's control node
  - Containerized runtime environment

```
$ ansible-navigator --version
ansible-navigator 2.1.0
```

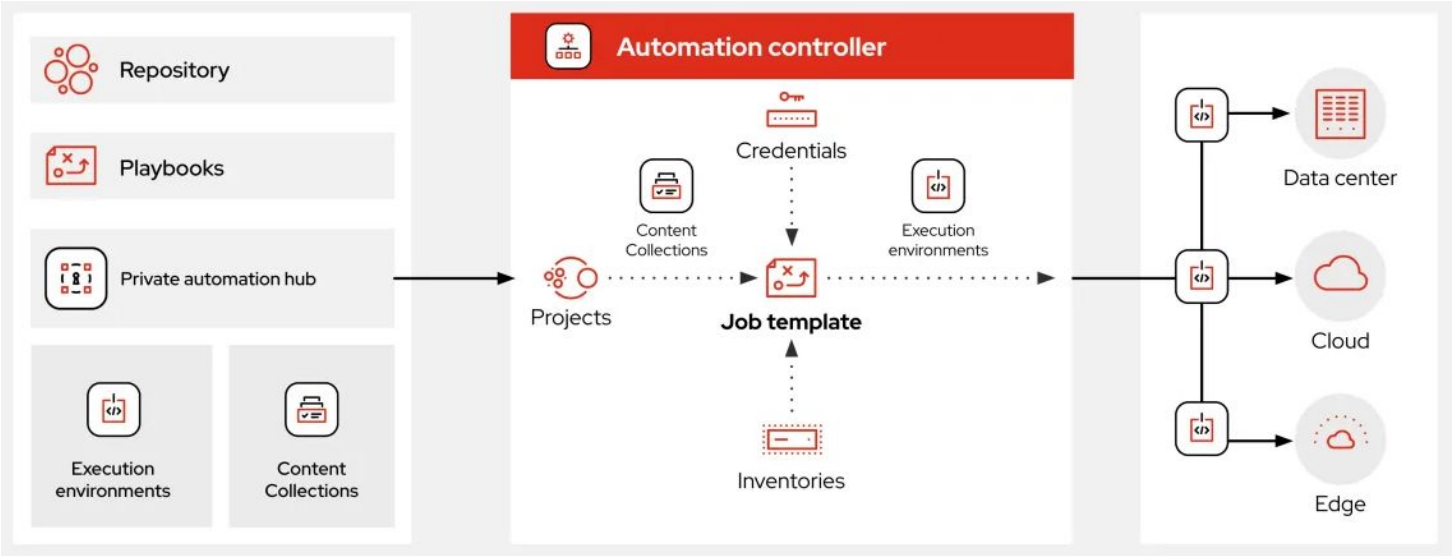
Ansible Execution Environment

- A container image that includes:
  - Ansible Core
  - Required Content Collections
  - Python libs, CLI tools, and dependencies
- Benefits:
  - Reproducible and portable execution
  - Seamless transition from dev to prod
- Used by:
  - ansible-navigator
  - Automation Controller

```
podman pull
registry.redhat.io/ansible-automation-platform-22/ee-supported-rhel8:latest
```

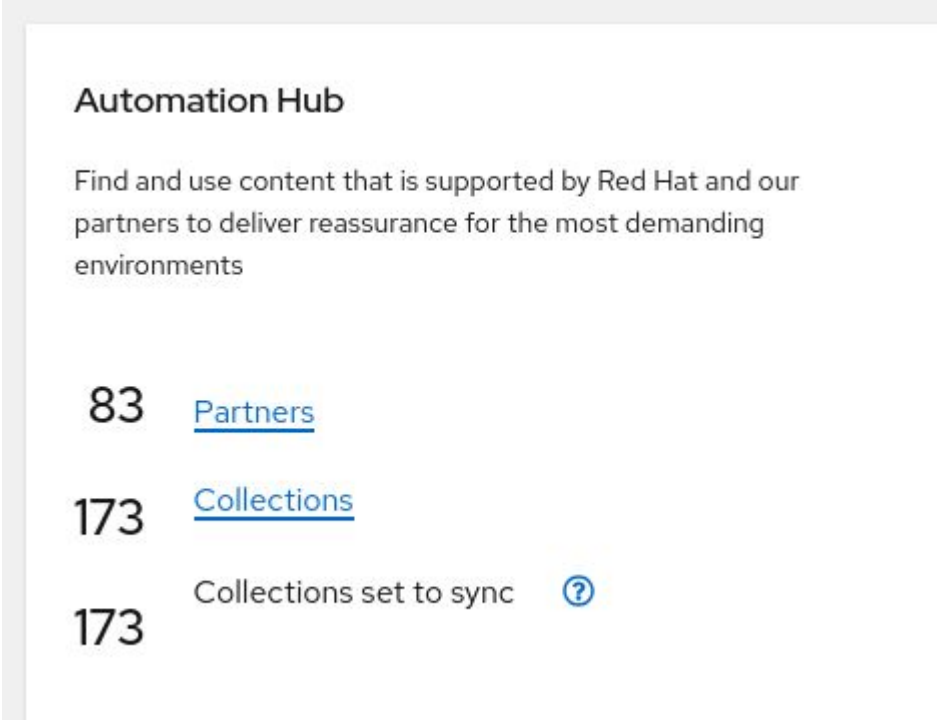
Ansible Automation Controller

- Formerly known as **Ansible Tower**
- Centralized Web UI and REST API for managing automation
- Key features:
  - Role-based access control (RBAC)
  - Job scheduling and logging
  - Credentials and inventory management



Ansible Automation Hub

- Hosted at console.redhat.com
- Provides certified content collections for Red Hat customers
- Can be integrated with:
  - ansible-galaxy
  - ansible-navigator
  - Automation Controller
- Ensures trusted, supported content for enterprise use



Ansible Inventory

What is an Ansible Inventory ?

- An inventory defines the hosts and groups of hosts that Ansible manages.
- It can be a simple INI file, YAML file, or a dynamic inventory (e.g., from cloud sources).
- 📁 Default file location: /etc/ansible/hosts (or specify via -i option)
- 🔧 Example (INI format):

```
[web]
servera.lab.example.com

[db]
192.168.1.20
```

Grouping Hosts

- Hosts can be organized into named groups.
- Groups can be nested using children.

```
[web]
web1 ansible_host=192.168.10.11
web2 ansible_host=192.168.10.12

[db]
db1 ansible_host=192.168.10.21

[production:children]
web
db
```

Hostnames expansion

- Hostnames can be written explicitly or expanded via range syntax.
- 🔧 Example of expansion:

```
[web]
web[01:03].example.com
```

- Expands to:
  - web01.example.com
  - web02.example.com
  - web03.example.com

Custom Host variables

- Each host can have specific variables, such as SSH port, user, or facts.
- 🔧 Example:

```
[web]
web1 ansible_host=10.1.1.1 ansible_user=admin ansible_port=2222
```

- These override settings in ansible.cfg.
- 🎯 Use host vars to fine-tune behavior without hardcoding into playbooks.

Inventory file setting

- You can choose which inventory file to use on the command line:

```
ansible-navigator run playbook.yml -i inventory_file
```

- Command line overrides the default setting in ansible.cfg

```
[defaults]
inventory = ./inventory.ini
```

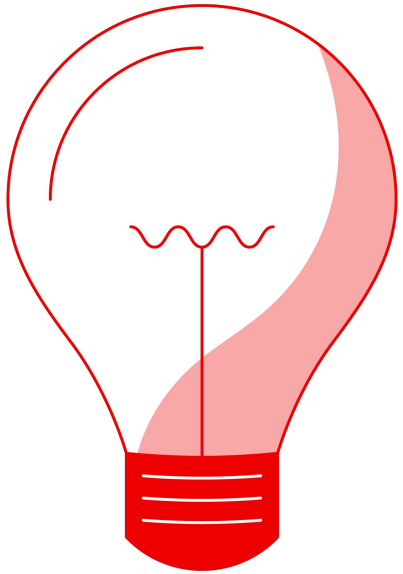
Dynamic Inventories

- Unlike static inventory files (.ini or .yaml), a dynamic inventory pulls host data at runtime from external sources like:
- Cloud providers (AWS, Azure, GCP)
- Custom scripts or plugins
- Example (script):

```
[defaults]
inventory = ./inventory_script.py
```

Ansible Inventory

Best practices



Group your hosts logically

Use groups like [db], [web], [loadbalancers] for clarity and reuse.

Use [children] for group hierarchies

Define groups of groups for layered infrastructure (e.g. prod includes db and web).

Use group_vars and host_vars directories

Store variables per group/host cleanly outside the inventory.

Prefer dynamic inventory for cloud/VMs

Use dynamic inventory scripts or plugins (e.g., AWS, OpenShift, Azure).

Source:
Insert source data here
Insert source data here

Introduction to Ansible Configuration

Ansible Configuration

Introduction to Ansible Configuration Files

Ansible relies on multiple configuration files to define behavior, control execution, and manage environments.

- Main configuration is in ansible.cfg
- Inventory, variables, and playbooks are separate but referenced
- Configuration can come from multiple sources: env vars, CLI, config files

```
# shows configuration settings
ansible --version
```

Ansible Configuration

ansible.cfg Configuration File

The ansible.cfg file controls global and project-specific settings for Ansible runs.

- Located in /etc/ansible/, ~/.ansible.cfg, or project root
- Defines inventory, remote user, roles path, timeouts, etc.
- Local ansible.cfg overrides system/global settings

```
[defaults]
inventory = ./inventory
remote_user = ansible
roles_path = ./roles
timeout = 10
```

Ansible Configuration

Configuring Privilege Escalation

Privilege escalation allows Ansible to run tasks as another user, typically root.

- Enabled via become: true in playbooks or CLI
- Controlled in ansible.cfg under [privilege_escalation]
- You can specify become_user, become_method, and password prompts

```
[privilege_escalation]
become = True
become_method = sudo
become_user = root
ask_become_pass = False
```

Ansible Configuration

ansible-navigator Configuration

ansible-navigator provides a terminal UI and YAML-based configuration for Ansible automation workflows.

- Uses ansible-navigator.yml file for settings
- Integrates with execution-environment (EE) containers
- Useful for consistent, containerized execution

```
# ansible-navigator.yml
execution-environment:
  image:
registry.redhat.io/ansible-automation-platform-22/ee-supported-rhel8:latest
  enabled: true

mode: stdout
playbook-artifact:
  enable: false
```

Ansible Playbooks

Ansible Playbooks

What are Playbooks?

Ansible Playbooks are YAML files that define a series of tasks to be executed on managed hosts.

- Written in YAML format
- Human-readable and machine-parsable
- Describe "what to do", not "how to do it"

Ansible Playbooks

Use Cases

Primary Use Cases:

- Configuration Management: Ensure systems are in a desired state
- Application Deployment: Deploy code and services across environments
- Provisioning: Create infrastructure components (e.g. VMs, containers)
- Orchestration: Coordinate complex multi-system workflows
- Security Automation: Enforce policies and deploy security patches
- Use across IT lifecycle, from dev to production

Ansible Playbooks

Playbook structure overview

- A playbook is made up of one or more plays, and each play targets a set of hosts.
- Main components:
  - hosts: Target group of systems
  - vars: Variables used in tasks
  - tasks: List of operations to perform
  - handlers: Triggered by notify when a change occurs
  - roles: Reusable collections of tasks, defaults, files
- Execution is ordered and deterministic.

Ansible Playbooks

Sample playbook

```
- name: Install and start Apache
  hosts: webservers
  tasks:
    - name: Install package
      ansible.builtin.yum:
        name: httpd
        state: present
    - name: Start service
      ansible.builtin.service:
        name: httpd
        state: started
```

Play name

Target Hosts

Tasks:

Module

Module

Ansible Playbooks

How to run a Playbook

```
ansible-playbook play.yml
```

- Runs playbook using local Python environment
- Uses locally installed collections and dependencies
- Requires all runtime tools (e.g., ansible, Python modules) to be installed on control node
- Simpler, but less portable and reproducible

```
ansible-navigator run -m stdout site.yml
```

- Runs playbook inside an Execution Environment (EE) container
- Automatically includes Ansible Core, collections, Python dependencies
- Ensures consistency across environments (dev/stage/prod)
- Preferred in Red Hat Ansible Automation Platform

Ansible Playbooks

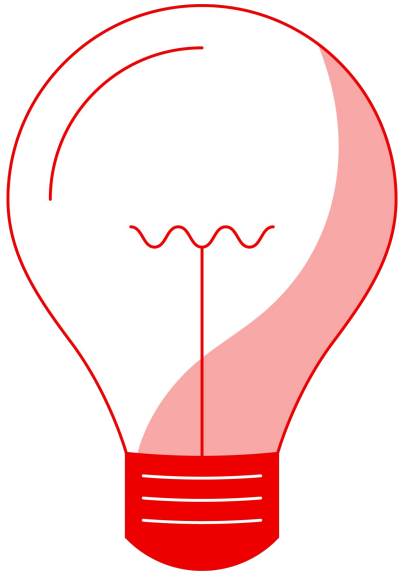
How to use multiple Plays in a Playbook

```
---
- name: Configure web server
  hosts: web
  become: yes
  tasks:
    - name: Install Apache
      yum:
        name: httpd
        state: present

- name: Configure database
  hosts: db
  become: yes
  tasks:
    - name: Start MariaDB
      service:
        name: mariadb
        state: started
```

- Every Play targets a set of hosts and uses its tasks
- Useful to run complementary activities
- Each Play has its own outcome (success, failure, etc)

Ansible Playbooks

Best practices



Name your tasks clearly

Always use the name: field in tasks.

Use --check and --syntax-check

--check lets you preview changes without applying them.

—-syntax-check verifies the syntax of commands

Avoid code duplication

Use include_tasks, import_tasks, or roles to reuse logic. (see you later!)

Document your variables

Add comments and defaults to variable files..Version control everything

Store playbooks, roles, inventories, and vars in Git.

Source:
Insert source data here
Insert source data here

Ansible Variables

Ansible Variables

What are Ansible Variables ?

- Key-value pairs used to store dynamic values in Ansible Playbooks.
- Simplify configuration management, reuse, and maintainability.
- Used for:
  - Host configurations
  - Package and service settings
  - Dynamic task decisions
  - Defined in multiple locations with precedence rules.

```
vars:
  user: joe
```

Ansible Variables

Ways to define Variables

- **Inventory variables**: Group and host variables in inventory or via group_vars and host_vars directories.
- **Playbook variables**: Using vars, vars_files, or vars_prompt.
- **Task variables**: Inline within a task for narrow scope.
- **Facts**: Automatically gathered about systems (ansible_facts).
- **Extra variables**: Passed on command line with -e (highest precedence).

📌 Tip: Use globally unique variable names to avoid collisions.

Variables Precedence

From Highest to Lowest

- 1 Extra vars (-e)
- 2 Task vars
- 3 Play vars / vars_files
- 4 Host facts
- 5 Host vars (host_vars)
- 6 Group vars (group_vars)
- 7 Inventory group vars
- 8 Role defaults
- 🎯 Narrow scope > Wide scope

```
# Example of highest precedence

ansible-navigator run site.yml -- -e "package=apache"
```

Ansible Variables in Playbooks

How to reference variables in Playbooks ?

- Use {{ variable_name }} for referencing:

```
tasks:
  - name: Create user {{ user }}
    user:
      name: "{{ user }}"
```

- Use variables in conditionals, loops, and module arguments.
- Always quote variables at the start of values to avoid YAML parsing issues.
- You can define variables externally:

```
- hosts: all
  vars_files:
    - vars/users.yml
```

Ansible Structured Variables

- Instead of using:

```
user1_first: Bob
user1_last: Jones
user1_home: /home/bjones
```

- Use:

```
users:
  bjones:
    first: Bob
    last: Jones
    home: /home/bjones
```

- Cleaner structure
- Easier management of related data
- Simplifies templating and loops.

Registering Variables

Capturing output

- Use register to store module or command outputs:

```
- name: Install httpd
  ansible.builtin.dnf:
    name: httpd
    state: installed
  register: install_result

- debug:
    var: install_result
```

- Useful for:
  - Conditional tasks based on command output.
  - Debugging module outputs.
  - Building dynamic workflows.

Ansible Variables

Best Practices

- ✅ Use host_vars and group_vars instead of inline inventory vars for clarity.
- ✅ Use consistent naming and avoid collisions.
- ✅ Prefer dictionaries for related variable sets.
- ✅ Keep variable values generic and reusable in Playbooks.
- ✅ Document variables and defaults clearly for your team.
- ✅ Use ansible-vault to encrypt sensitive variables.

Ansible Variables

Host Variables and Group Variables

Ansible allows assigning specific variables to individual hosts or groups of hosts in the inventory to customize configuration and behavior.

- Host variables apply to a single host.
- Group variables apply to all members of a group.
- Variables are defined in the inventory file or in separate YAML files.
- Ansible merges variables from multiple sources following precedence rules

```
# Inventory example
[web]
web1 ansible_host=192.168.1.10 http_port=80
web2 ansible_host=192.168.1.11 http_port=8080
```

Ansible Variables

Using Directories to Populate Host and Group Variables

Organize host and group variables in structured directories to improve readability and reusability.

- Use host_vars/ and group_vars/ directories in your Ansible project.
- Filenames should match hostnames or group names.
- Files must be in YAML format (.yml) or .json.

```
# File: host_vars/web1.yml
http_port: 80

# File: group_vars/web.yml
firewall_enabled: true
```

Ansible Variables

Overriding Variables from the Command Line

Command-line options let you override variable values during playbook execution for temporary customization.

- Use -e or --extra-vars to define overrides.
- Accepts key=value or YAML/JSON strings.
- Useful for passing secrets or environment-specific values.

```
ansible-navigator site.yml -e "http_port=8081"
```

Ansible Vault

Ansible Secrets

Introducing Ansible Vault

Ansible Vault allows you to securely store sensitive data like passwords, API tokens, and private keys in encrypted files.

- Useful for keeping secrets out of version control.
- Encryption is file-based and uses AES256.
- Vaulted files can be playbooks, vars files, or any text-based content.
- Decryption occurs automatically during playbook execution when the vault password is provided.

```
ansible-navigator run site.yml --vault-password-file ~/.vault_pass.txt
```

Ansible Secrets

Creating, Viewing, Editing, Encrypting, and Decrypting a File

You can manage Vault-protected files directly using Ansible Vault commands.

- Create new encrypted files with ansible-vault create.
- Edit encrypted files without decrypting them manually.
- View or decrypt files for debugging or migration.

```
ansible-vault create secrets.yml
ansible-vault view secrets.yml
ansible-vault edit secrets.yml
ansible-vault encrypt plain-vars.yml
ansible-vault decrypt secrets.yml
```

Ansible Secrets

Using Secrets in Playbooks

Vault-encrypted variables can be used seamlessly in playbooks just like plaintext variables.

- Include vaulted vars files in your vars_files.
- Ensure the vault password is provided at runtime.
- Supports multiple vault-encrypted files

```
vars_files:
  - secrets.yml
```

```
ansible-navigator run deploy.yml --vault-password-file ~/.vault_pass.txt
```

Ansible Secrets

Recommended Practices for Variable File Management

To securely manage variables in larger projects, follow these best practices.

- Store secrets in group_vars/ or host_vars/ with Vault encryption.
- Avoid placing secrets directly inside the playbook.
- Use --vault-id to manage multiple vault passwords if needed.
- Keep the vault password file protected and outside version control

```
ansible-navigator run site.yml --vault-id prod@~/.vault_pass.txt
```

Ansible Secrets

Encrypting Specific Variables Inline

You can encrypt only specific variables inside a file using ansible-vault encrypt_string.

- Useful when only part of a vars file is sensitive.
- Can be pasted directly into playbooks or YAML files.
- Supports tags and comments for documentation.

```
ansible-vault encrypt_string 'mysecretpassword' --name 'db_password'
```

```
db_password: !vault |
          $ANSIBLE_VAULT;1.1;AES256...
```

Ansible Facts

Ansible Facts

What are Ansible Facts ?

Ansible facts are system properties automatically discovered from managed nodes and used to make playbooks dynamic.

- Collected at the beginning of each play by default
- Includes information like IP, OS, memory, and more
- Accessible as variables in tasks and templates
- Uses the setup module behind the scenes

```
# When you run the playbook, the facts are displayed in the job output
ansible-navigator run playbook.yml --mode stdout
```

Ansible Facts

Ansible Facts Injected as Variables

Facts are injected as host-level variables and can be used like any other variable in Ansible tasks.

- Syntax: ansible_facts['key'] or shorthand ansible_key
- Available automatically after fact gathering
- Useful in conditionals, templates, and debug output

```
- name: Show distribution
  debug:
    msg: "Running on {{ ansible_distribution }} {{
ansible_distribution_version }}"
```

Ansible Facts

Turn off fact gathering

Fact gathering can be disabled to speed up execution when facts are not needed.

- Use gather_facts: false in the play definition
- Reduces playbook runtime
- Can also be skipped per host or conditionally

```
- name: Run without gathering facts
  hosts: all
  gather_facts: false
```

Ansible Facts

Gathering only a subset of Facts

Gathering only the facts you need reduces overhead and output size.

- Use setup with gather_subset option
- Examples: hardware, network, virtual
- Can be run manually in specific tasks

```
- name: Gather only network and hardware facts
  setup:
    gather_subset:
      - network
      - hardware
```

Ansible Facts

Creating Custom Facts

Custom facts add user-defined data to your fact set.

- Stored as .ini, .json, or .yaml in /etc/ansible/facts.d/
- Loaded automatically with standard facts
- Good for roles, environment labels, or metadata

```
- name: Copy custom fact file
  copy:
    src: my_custom.fact
    dest: /etc/ansible/facts.d/my_custom.fact
    mode: '0644'
```

```
[general]
role=webserver
location=datacenter1
```

Ansible Facts

Creating Facts from other variables

Use **set_fact** to define new variables dynamically during playbook execution.

- Combine or transform existing variables
- Useful for building file paths, URLs, or logic controls
- Variables persist for the duration of the play

```
- name: Set a derived variable
  set_fact:
    full_hostname: "{{ inventory_hostname }}.{{ dns_suffix }}"
```

Ansible Facts

Using Magic Variables

Magic variables provide execution context and inventory data.

- inventory_hostname, group_names, hostvars, etc.
- Allow referencing data from other hosts or groups
- Essential for dynamic roles and multi-host coordination

```
- name: Show IP of another host
  debug:
    msg: "DB server IP is {{ hostvars['db01'].ansible_default_ipv4.address
}}"
```

Ansible Task Control

Ansible Task Control

Implementing Task Control

Control how and when tasks are executed in a playbook.

- Use when to run tasks conditionally
- Use block, rescue, and always for structured error handling
- Control handlers execution with notify
- Set task retries with retries and delay

```
tasks:

  - name: Controllo condizionale con when
    command: echo "Solo se serve"
    when: ansible_hostname == "localhost"
```

Ansible Task Control

Implementing Task Control

Example (continued):

```
tasks:

  - name: Retry su servizio HTTP instabile
     uri:
       url: http://localhost:8080/status
       return_content: yes
     retries: 3
     delay: 5
     until: http_result.status == 200

   - name: Block con rescue
     block:
       - name: Task che fallisce
         command: /bin/false
     rescue:
       - name: Gestione dell’errore
         debug:
           msg: "⚠️ Errore gestito senza fermare il playbook"
```

Ansible Task Control

Writing Loops and Conditional Tasks (Simple Loops)

Repeat a task over a list of items.

- Use loop or the legacy with_items
- Each iteration runs the same task with a different value
- item refers to the current element in the loop

```
- name: Install packages
  ansible.builtin.yum:
    name: "{{ item }}"
    state: present
  loop:
    - httpd
    - php
    - mariadb-server
```

Ansible Task Control

Writing Loops over a List of Dictionaries

Access multiple values in each loop item

- Loop over complex structures like dictionaries
- Use item.key notation to access dictionary values
- Useful for templating or multi-property configuration

```
- name: Create users
  ansible.builtin.user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'alice', groups: 'dev' }
    - { name: 'bob', groups: 'ops' }
```

Ansible Task Control

Using Register Variables with Loops

Capture loop results for later use

- Use register to save task output results key holds each iteration result
- Combine with debug or conditionals

```
- name: Ping multiple hosts
  ansible.builtin.shell: "ping -c1 {{ item }}"
  loop:
    - server1
    - server2
  register: ping_results
```

Ansible Task Control

Running Tasks Conditionally

Control task execution using variables or facts

- Use when with booleans, strings, or expressions
- Combine with ansible_facts or output from register
- Condition must evaluate to true

```
- name: Restart webserver if updated
  ansible.builtin.service:
    name: httpd
    state: restarted
  when: webserver_updated.changed
```

Ansible Task Control

Testing Multiple Conditions

Use logical operators to combine conditions

- Combine expressions with and, or, not
- Brackets recommended for clarity
- Can mix facts, variables, and register outputs

```
- name: Run only on RedHat with httpd installed
  ansible.builtin.debug:
    msg: "All conditions met"
  when: (ansible_facts['os_family'] == "RedHat") and (httpd_check.rc == 0)
```

Ansible Task Control

Combining Loops and Conditional Tasks

Skip loop items based on their values

- Use when inside a loop to filter specific items
- item still available for conditional evaluation
- Improves performance and avoids unwanted changes

```
- name: Delete old users
  ansible.builtin.user:
    name: "{{ item }}"
    state: absent
  loop:
    - testuser
    - admin
  when: item != "admin"
```

Ansible Handlers

Ansible Handlers

What are Ansible Handlers?

Handlers are special tasks triggered only when notified by another task

- Used for actions that should happen only if changes occur
- Typical use cases: restarting services, reloading configurations
- Declared just like regular tasks, but referenced using notify

```
- name: Install nginx
  ansible.builtin.yum:
    name: nginx
    state: latest
  notify: Restart nginx
```

Ansible Handlers

Implementing Handlers

Define and organize handlers for event-driven automation

- Handlers are defined under the handlers section in a playbook or role
- Triggered only once per play, even if notified multiple times
- Good practice: group all handlers at the end of the playbook

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

Ansible Handlers

Benefits of using Handlers

Handlers improve efficiency, clarity, and control in automation

- Prevent unnecessary service restarts or reloads
- Make playbooks more idempotent and performance-oriented
- Reduce noise in logs by executing only on change

```
# Service restarts only if configuration has changed
ansible-navigator run webserver.yml
```

Handling Task Failures

Handling Failures

How to handle Task failures in Ansible

Ansible tasks may fail due to unreachable hosts, invalid parameters, or runtime errors

- By default, play execution stops on the first failed task
- Use ignore_errors: true to continue execution
- Failures are still logged and reported

```
- name: Try to install a package
  ansible.builtin.yum:
    name: unknown-package
    state: present
  ignore_errors: true
```

Handling Failures

Managing Task Errors in Plays

Control the flow of execution after failures using rescue and always blocks

- Use **block** to group tasks
- Use **rescue** to define fallback tasks if a block fails
- Use **always** for tasks that must run regardless of success or failure

```
- block:
    - name: Try risky task
      command: /fail-prone-script.sh
  rescue:
    - name: Run fallback
      debug:
        msg: "Task failed, fallback executed"
  always:
    - name: Always run cleanup
      file:
        path: /tmp/tempfile
        state: absent
```

Handling Failures

Specifying Task Failure Conditions

Force a task to fail based on logic or command result

- Use failed_when to set a custom failure condition
- Allows more precise error detection based on output
- Useful with register for evaluating results

```
- name: Check if status code is not 200
  uri:
    url: http://example.com
    return_content: no
  register: result
  failed_when: result.status != 200
```

Handling Failures

Specifying When a Task Reports "Changed" Results

Control whether a task is considered “changed” or not

- Use changed_when to define a condition for marking the task as changed
- Helps avoid false positives in change tracking
- Combine with check_mode or register for accuracy

```
- name: Run script only if config is outdated
  command: ./update-config.sh
  register: update_output
  changed_when: "'updated' in update_output.stdout"
```

Handling Failures

Ansible Blocks and Error Handling

Blocks allow logical grouping of tasks and provide structure to control their execution.

- Use block to group related tasks under a shared condition
- Apply when, become, tags at the block level
- Combine block, rescue, and always for advanced error recovery

```
tasks:
  - block:
      - name: Install package
        yum:
          name: httpd
          state: present
      - name: Start service
        service:
          name: httpd
          state: started
    when: ansible_facts['os_family'] == "RedHat"
```

Modifying and Copying Files

Modify and Copy Files on Managed Hosts

Deploying Files to Managed Hosts

Transferring files from the control node to managed hosts is a common and essential automation task in configuration management.

- Use copy to transfer static files
- template for Jinja2-based dynamic file generation
- Common use: config files, scripts, environment definitions
- Ensure files are idempotently deployed

```
- name: Deploy a configuration file
  ansible.builtin.copy:
    src: files/myconfig.conf
    dest: /etc/myapp/config.conf
    owner: root
    group: root
    mode: '0644'
```

Modify and Copy Files on Managed Hosts

Modifying and Copying Files to Hosts (ansible.builtin)

Ansible provides several built-in modules for Linux file management, such as creating, modifying, or transferring files. copy: Transfer files from control node to managed hosts, with attribute control

- **file**: Set permissions, ownership, or create/remove files, links, and directories
- **lineinfile**: Ensure or modify a single line inside a file
- **blockinfile**: Insert, update, or remove multiline blocks with markers
- **fetch**: Copy files from managed hosts to the control node
- **stat**: Check file status and metadata (like ls -l or stat)
- **synchronize**: Efficient file tree sync using rsync (from ansible.posix)
- **patch**: Apply GNU patch files to managed content (from ansible.posix)

Modify and Copy Files on Managed Hosts

Automation Examples with Files Modules

File-related modules can be combined to implement repeatable system configurations.

- Configure files dynamically using facts or variables
- Combine lineinfile and blockinfile for in-place editing
- Use stat to conditionally apply file actions
- Automate compliance tasks or fix misconfigurations

```
- name: Ensure setting is present in config
  ansible.builtin.lineinfile:
    path: /etc/sysctl.conf
    regexp: '^net.ipv4.ip_forward'
    line: 'net.ipv4.ip_forward = 1'
```

Modify and Copy Files on Managed Hosts

Modifying File Attributes

The file module allows you to manage permissions, ownership, and symbolic links.

- Set permissions with mode, owner, and group
- Supports “state” to enforce the status
- Create/remove directories and symlinks
- Enforce consistent file system state

```
- name: Ensure the /opt/myapp directory exists
  ansible.builtin.file:
    path: /opt/myapp
    state: directory
    owner: myuser
    group: mygroup
    mode: '0755'
```

Modify and Copy Files on Managed Hosts

Copying / Editing / Removing Files on Managed Hosts

Ansible makes it easy to copy, edit, and delete files across systems.

- Use copy and template to create files
- Use lineinfile, blockinfile for inline edits
- Use file with state: absent to remove files or directories
- Files can be conditionally managed with when or stat

```
- name: Remove old backup file
  ansible.builtin.file:
    path: /etc/myapp/old.conf.bak
    state: absent
```

Modify and Copy Files on Managed Hosts

Synchronizing Files Between the Control Node and Managed Hosts

The synchronize module wraps rsync to perform efficient file transfers..

- Use for large files or recursive directory sync
- More performant than copy for big payloads
- Can preserve permissions, delete missing files
- Requires rsync to be available on both ends

```
- name: synchronize local file to remote files
  ansible.posix.synchronize:
    src: file
    dest: /path/to/file
```

Jinja2 Templates

Jinja2 Templates

Introduction to Jinja2 Templates

Jinja2 is a powerful templating engine used in Ansible to generate dynamic content in files and playbooks.

- Allows embedding variables, loops, and conditionals in files
- Commonly used for config files, scripts, or documents
- Template files typically use the .j2 extension
- Evaluated at runtime using Ansible facts and variables

Jinja2 Templates

Building a Jinja2 Template

A Jinja2 template contains plain text with embedded variables or logic that Ansible renders dynamically.

- Use {{ variable }} to insert values
- Support for loops {% for %} and conditions {% if %}
- Template files stored separately in the templates/ directory

```
Hello, my name is {{ username }} and I manage {{ server_count }} servers.
```

Jinja2 Templates

Deploying a Jinja2 Template

Templates are deployed using the template module in an Ansible task.

- Source is a .j2 file under templates/
- Destination is the path on the managed node
- All variables used must be defined in inventory or playbook

```
- name: Deploy nginx config
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/nginx.conf
```

Jinja2 Templates

Managing Template files

Templates should be organized for maintainability and reuse.

- Place all .j2 files in the **templates**/ directory
- Use variables to customize for different environments
- Use defaults/ or vars/ to define common template values
- Use includes or partials to reduce duplication

```
tree templates
├── index.html.j2
├── nginx.conf.j2
└── partials
    └── vhost.j2
```

Jinja2 Templates

Managing Loops in Template files

Loops allow generating repetitive content such as lists, entries, or blocks.

- Use {% for item in list %} to start a loop
- Always close with {% endfor %}
- Useful for generating lines from variable arrays

```
{% for user in users %}
User: {{ user.name }}, Role: {{ user.role }}
{% endfor %}
```

Jinja2 Templates

Using Conditionals in Template files

Conditionals control rendering based on variable values.

- Use {% if condition %} and {% else %} blocks
- Close with {% endif %}
- Supports is defined, equality, and boolean logic

```
{% if enable_ssl %}
SSL is enabled on {{ domain }}
{% else %}
Running without SSL
{% endif %}
```

Jinja2 Templates

Variable filters in templates

Filters transform or format variables in templates.

- Syntax: {{ variable | filter_name }}
- Common filters: default, upper, lower, join, replace, length
- Helps control formatting and fallbacks

```
User: {{ username | default("guest") | upper }}
List: {{ items | join(", ") }}
```

Selecting Hosts with Patterns

Referencing Inventory Hosts

How to reference Inventory Hosts with Host Patterns

You can select which hosts Ansible will manage by using host patterns in your playbooks, matching hosts listed in your inventory

- Use the exact host name as listed in your inventory
- Patterns determine which hosts Ansible will target
- Allows granular control over your playbooks

```
[all]
webserver1 ansible_host=192.168.1.10
```

```
- hosts: webserver1
  tasks:
    - name: Test connectivity
      ping:
```

Referencing Inventory Hosts

Specifying Hosts Using a Group

You can organize hosts into groups within your inventory file and target them as a single unit.

- Define groups in your inventory using brackets
- Execute tasks on all hosts within a group simultaneously
- Useful for roles and environment-based grouping

```
[webservers]
web1 ansible_host=192.168.1.11
web2 ansible_host=192.168.1.12
```

```
- hosts: webservers
  tasks:
    - name: Test connectivity
      ping:
```

Referencing Inventory Hosts

Matching Multiple Hosts with Wildcards

You can use wildcard patterns to match multiple hosts dynamically within your inventory.

- Use * to match zero or more characters
- Helps when hostnames follow consistent naming patterns
- Provides flexible targeting without explicit group creation

```
[webservers]
web1 ansible_host=192.168.1.11
web2 ansible_host=192.168.1.12
```

```
- hosts: "web*"
  tasks:
    - name: Test connectivity
      ping:
```

Referencing Inventory Hosts

Referencing Multiple entries in an inventory using logical lists

You can combine multiple host patterns using logical operations to target specific subsets of your inventory.

- Use : to separate multiple groups or hosts (OR)
- Use & to intersect groups (AND)
- Use ! to exclude hosts or groups (NOT)

```
[webservers]
web1 ansible_host=192.168.1.11
web2 ansible_host=192.168.1.12

[dbservers]
db1 ansible_host=192.168.1.21
```

```
- hosts: "webservers:dbservers"
  tasks:
    - name: Test connectivity
```

Including and Importing Files

Import and Include Files

Including and Importing Playbook and Task Files

By using imports and includes you can reuse common task or play definitions across multiple playbooks

- Improve modularity and readability of Ansible code
- Ansible provides both include and import statements for this purpose
- Behavior differs based on when the file is processed (parse time vs runtime)

```
- hosts: webservers
  tasks:
    - import_tasks: common_setup.yml
    - include_tasks: deploy_app.yml
```

Import and Include Files

Importing Files in Ansible

- Files are imported at parse time
- All imported tasks are loaded before execution begins
- Cannot be used with when to conditionally load different files
- Better for static logic and when control flow is not required

```
- hosts: all
  tasks:
    - import_tasks: users.yml
```

Import and Include Files

Including Files in Ansible

- Files are included at runtime
- Allows dynamic inclusion based on conditions (e.g., when)
- Useful when you need logic to decide which file to include
- Can be used inside a block or role

```
- hosts: all
  tasks:
    - name: Include OS-specific tasks
      include_tasks: "tasks/{{ ansible_os_family }}.yml"
```

Import and Include Files

Key Differences Between Including and Importing

- include: evaluated at **runtime**, supports conditionals like when
- import: evaluated at **parse time**, does not support conditional logic
- Imports are statically known before execution begins
- Includes provide more flexibility but less predictability
- Use imports for predictable structure, includes for dynamic behavior

```
# Can conditionally include tasks
- include_tasks: tasks/setup.yml
  when: install_required

# Applies the condition to all tasks (cannot conditionally include)
- import_tasks: tasks/setup.yml
  when: install_required
```

Ansible Roles And Collections

Ansible Roles and Collections

Ansible Roles: Purpose and Benefits

Roles provide a modular structure to organize and reuse Ansible code effectively.

- Encapsulate tasks, handlers, variables, and templates
- Promote reuse and separation of concerns
- Easier to read, maintain, and test
- Can be shared or downloaded from external sources

```
roles:
  - webserver
```

Ansible Roles and Collections

Ansible Role Directory Structure

Roles follow a specific directory layout to organize related resources.

- **tasks**: Main list of tasks
- **handlers**: Handlers triggered by notify
- **vars, defaults**: Variables with different precedence
- **templates, files**: Jinja2 templates and static files
- **meta**: Role dependencies

```
tree roles/rolename/
```

Ansible Roles and Collections

Creating a Role in a Project

You can create a new role manually or with **ansible-galaxy** init.

- Define a role for reusable functionality
- Include it in playbooks with the roles keyword
- Customize using role variables

```
ansible-galaxy init webserver
```

Ansible Roles and Collections

Using a Role in a Playbook

Include roles in your playbook to apply their logic to target hosts.

- Defined under the roles: section of a play
- Automatically executes main.yml in tasks/
- Simplifies playbook logic

```
- hosts: all
  roles:
    - webserver
```

Ansible Roles and Collections

Different Ways to Use Roles in Plays

Roles can be applied using the roles keyword or using include_role/import_role.

- roles: is declarative and used directly in plays
- include_role: is dynamic and can be conditional
- import_role: is static and processed at parse time

```
- hosts: all
  tasks:
    - name: Dynamically include a role
      include_role:
        name: webserver
      when: ansible_os_family == "RedHat"
```

Ansible Roles and Collections

Retrieving Roles from external sources

External roles can be cloned from external sources like Git repository.

- Use requirements.yml to define sources
- Supports Git, Galaxy, and local paths
- Useful for internal or shared repositories

```
- src: https://github.com/example/web-role.git
  name: webserver
```

```
ansible-galaxy install -r requirements.yml
```

Ansible Roles and Collections

Installing Roles from Ansible Galaxy

Ansible Galaxy is the central hub for community roles.

- Explore roles at https://galaxy.ansible.com
- Use ansible-galaxy or requirements.yml to install roles
- Installed to ~/.ansible/roles by default

```
ansible-galaxy install geerlingguy.apache
```

Ansible Roles and Collections

What Are Ansible Content Collections?

Collections bundle roles, modules, and plugins into a distributable package.

- Organize content under namespaces
- Preferred distribution model over standalone roles
- Useful for vendor-supported automation (Red Hat supported Roles are distributed as part of a Collection)

```
ansible-galaxy collection install community.mysql
```

Ansible Roles and Collections

Using a Role from a Collection

Access roles and modules from a collection using the fully qualified name.

- Namespace and collection prefix required
- Ensures proper content isolation
- Collections installed in /usr/share/ansible/collections or similar

```
- hosts: db_servers
  roles:
    - role: community.mysql.mysql_install  # Collection
```

```
- hosts: db_servers
  roles:
    - role: mysql_install  # Local Role
```

Ansible Roles and Collections

Common Command-Line Options for Roles and Collections

Use CLI tools to manage, install, and inspect roles and collections.

- **ansible-galaxy install**: Install roles or collections
- **ansible-galaxy list**: List installed content
- **ansible-galaxy init**: Create a new role structure
- **ansible-galaxy collection install** <name>: Install a collection

```
ansible-galaxy role list
ansible-galaxy collection install ansible.posix
ansible-galaxy init myrole
```

Ansible Roles and Collections

When to Use Roles vs When to Use Collections

Choose the right abstraction depending on scope, reuse, and distribution needs.

- Use roles for modular logic within a project
- Use collections for packaging and distributing multiple roles, modules, plugins
- Collections are ideal for shared, vendor-maintained content
- Roles are lighter and easier to develop locally

Ansible Automation Platform

Thank you!

linkedin.com/company/red-hat

youtube.com/user/RedHatVideos

facebook.com/redhatinc

x.com/RedHat

Red Hat