URADICAL

POCKET REFERENCE

ansible

Infrastructure as Code

Alan Bradley

Contents

Chapter 1

About This Book

Ansible is an open-source automation platform that makes it easy to configure systems, deploy software, and orchestrate complex IT workflows. Unlike other configuration management tools, Ansible is agentless—it connects to managed nodes using SSH and requires no special software to be installed on target systems.

This pocket reference provides a concise yet thorough overview of Ansible's core features, syntax, and best practices. Whether you're writing your first playbook or looking up advanced module options, this guide will help you work more efficiently with Ansible.

Who This Book Is For:

- System administrators automating infrastructure
- DevOps engineers building CI/CD pipelines
- Developers managing application deployments
- Anyone learning infrastructure as code

Chapter 2

Installation

Ansible can be installed on most Unix-like systems. The control node (where Ansible runs) requires Python 3.9 or later. Managed nodes only need SSH access and Python.

Recommended Installation Method:

The pip installation provides the latest version and works across all platforms:

```
# pip (recommended)
pip install ansible
```

Platform-Specific Installation:

```
# macOS
brew install ansible

# Ubuntu/Debian
sudo apt install ansible

# RHEL/CentOS/Fedora
sudo dnf install ansible
```

Verifying Installation:

```
# Check version and configuration
ansible --version
```

This displays the Ansible version, configuration file location, configured module search paths, and Python version being used.

Chapter 3

Core Concepts

Understanding these fundamental concepts is essential for working effectively with Ansible.

Concept	Description
Control Node	The machine where Ansible is installed and runs from. This is typically your workstation or a dedicated automation server.
Managed Node	Target hosts that Ansible configures. These are the servers, network devices, or cloud resources you're automating.
Inventory	A list of managed nodes organized into groups. Can be a simple text file or dynamically generated from cloud providers.
Playbook	A YAML file containing a series of plays that define automation workflows. Playbooks are Ansible's configuration, deployment, and orchestration language.
Task	A single unit of work that calls an Ansible module. Tasks are executed sequentially within a play.
Module	Reusable code units that perform specific actions like installing packages, copying files, or managing services. Ansible includes thousands of built-in modules.
Role	A structured way to organize related tasks, variables, files, and templates into reusable components. Roles promote code reuse and maintainability.
Collection	A distribution format for Ansible content including roles, modules, and plugins. Collections are shared via Ansible Galaxy.

How Ansible Works:

1. You write playbooks describing the desired state of your systems
2. Ansible reads the inventory to determine which hosts to manage
3. Ansible connects to managed nodes via SSH (or other connection methods)
4. Modules are transferred and executed on the remote systems
5. Results are collected and reported back to the control node

Chapter 4

Directory Structure

A well-organized project structure makes your Ansible code easier to maintain and collaborate on. Here's the recommended layout:

```
project/
+-- ansible.cfg           # Project config
+-- inventory/
|   +-- production        # Production hosts
|   +-- staging           # Staging hosts
+-- group_vars/
|   +-- all.yml           # Variables for all groups
|   +-- webservers.yml    # Variables for webservers group
+-- host_vars/
|   +-- web1.example.com.yml
+-- roles/
|   +-- common/
|       +-- tasks/
|       +-- handlers/
|       +-- templates/
|       +-- files/
|       +-- vars/
|       +-- defaults/
|       +-- meta/
+-- playbooks/
|   +-- site.yml
+-- requirements.yml      # Role/collection dependencies
```

Key Directories Explained:

- **inventory/** — Contains host definitions organized by environment
- **group_vars/** — Variables automatically loaded for host groups
- **host_vars/** — Variables specific to individual hosts
- **roles/** — Reusable automation components
- **playbooks/** — Main playbook files

Chapter 5

Configuration (ansible.cfg)

Ansible's behavior can be customized through configuration files. Settings are loaded in this order (first found wins):

1. `ANSIBLE_CONFIG` environment variable
2. `./ansible.cfg` in the current directory
3. `~/.ansible.cfg` in the user's home directory
4. `/etc/ansible/ansible.cfg` system-wide defaults

Common Configuration Options:

```
[defaults]
inventory = ./inventory/hosts
remote_user = deploy
private_key_file = ~/.ssh/id_ed25519
host_key_checking = False
retry_files_enabled = False
gathering = smart
fact_caching = jsonfile
fact_caching_connection = /tmp/ansible_facts
fact_caching_timeout = 86400

[privilege_escalation]
become = True
become_method = sudo
become_user = root
become_ask_pass = False

[ssh_connection]
pipelining = True
control_path = /tmp/ansible-%%h-%%p-%%r
```

Configuration Highlights:

- **inventory** — Default inventory file location
- **remote_user** — SSH user for connections
- **gathering** — When to collect system facts (smart caches facts)
- **pipelining** — Reduces SSH operations for better performance
- **become** — Enable privilege escalation by default

Chapter 6

Inventory

The inventory defines which hosts Ansible manages and how they're organized. Ansible supports both static files and dynamic inventory scripts.

INI Format

The traditional format uses INI-style syntax:

```
# Simple hosts
web1.example.com
web2.example.com

# Groups
[webservers]
web1.example.com
web2.example.com ansible_port=2222

[dbservers]
db1.example.com ansible_user=postgres
db2.example.com

# Group of groups
[production:children]
webservers
dbservers

# Group variables
[webservers:vars]
http_port=80
```

YAML Format

YAML provides a more structured alternative:

```
all:
  children:
    webservers:
      hosts:
        web1.example.com:
        web2.example.com:
          ansible_port: 2222
      vars:
        http_port: 80
    dbservers:
      hosts:
        db1.example.com:
          ansible_user: postgres
        db2.example.com:
    production:
      children:
        webservers:
        dbservers:
```

Dynamic Inventory

For cloud environments, dynamic inventory automatically discovers hosts:

```
# AWS EC2
pip install boto3
ansible-inventory -i aws_ec2.yml --list

# aws_ec2.yml
plugin: amazon.aws.aws_ec2
regions:
  - eu-west-1
keyed_groups:
```

```
  - key: tags.Environment
    prefix: env
```

Connection Variables

These variables control how Ansible connects to hosts:

Variable	Description
`ansible_host`	IP or hostname to connect to
`ansible_port`	SSH port (default: 22)
`ansible_user`	SSH username
`ansible_password`	SSH password (use vault)
`ansible_ssh_private_key_file`	SSH key path
`ansible_connection`	Connection type: ssh, local, docker
`ansible_python_interpreter`	Path to Python on target
`ansible_become`	Enable privilege escalation
`ansible_become_method`	sudo, su, pbrun, etc.
`ansible_become_user`	User to become

Chapter 7

Ad-Hoc Commands

Ad-hoc commands let you run quick, one-off tasks without writing a playbook. They're perfect for simple operations, testing, and exploration.

Basic Syntax:

```
ansible <pattern> -m <module> -a "<arguments>" [options]
```

Common Examples:

```
# Ping all hosts
ansible all -m ping

# Run command
ansible webservers -m command -a "uptime"
ansible webservers -m shell -a "cat /etc/passwd | grep root"

# Copy file
ansible webservers -m copy -a "src=/local/file dest=/remote/file"

# Install package
ansible webservers -m apt -a "name=nginx state=present" -b

# Manage service
ansible webservers -m service -a "name=nginx state=started enabled=yes" -b

# Gather facts
ansible web1 -m setup
ansible web1 -m setup -a "filter=ansible_distribution*"

# Check mode (dry run)
ansible webservers -m apt -a "name=nginx state=present" -b --check

# Limit to specific hosts
ansible webservers -m ping --limit "web1.example.com"

# Parallel execution
ansible all -m ping -f 10  # 10 forks
```

Host Patterns

Patterns determine which hosts a command targets:

Pattern	Matches
`all` or `*`	All hosts in inventory
`webservers`	Hosts in the webservers group
`webservers:dbservers`	Union of groups (OR)
`webservers:&production`	Intersection of groups (AND)
`webservers:!staging`	Exclusion (NOT)
`web*.example.com`	Wildcard matching
`~web[0-9]+`	Regular expression

Chapter 8

Playbooks

Playbooks are Ansible's configuration, deployment, and orchestration language. They describe the desired state of your systems using YAML syntax.

Basic Structure

A playbook contains one or more plays, each targeting a set of hosts:

```
---
- name: Configure webservers
  hosts: webservers
  become: yes
  gather_facts: yes
  vars:
    http_port: 80

  pre_tasks:
    - name: Update apt cache
      apt:
        update_cache: yes
        cache_valid_time: 3600

  tasks:
    - name: Install nginx
      apt:
        name: nginx
        state: present

    - name: Start nginx
      service:
        name: nginx
        state: started
        enabled: yes

  handlers:
    - name: Restart nginx
      service:
        name: nginx
        state: restarted

  post_tasks:
    - name: Verify nginx is running
      uri:
        url: "http://localhost:{{ http_port }}"
        status_code: 200
```

Play Components:

- **name** — Human-readable description
- **hosts** — Target hosts or groups
- **become** — Enable privilege escalation
- **gather_facts** — Collect system information
- **vars** — Play-level variables
- **pre_tasks** — Tasks that run before roles
- **tasks** — Main task list
- **handlers** — Tasks triggered by notifications
- **post_tasks** — Tasks that run after roles

Running Playbooks

```
# Basic run
ansible-playbook playbook.yml

# Specify inventory
ansible-playbook -i inventory/production playbook.yml

# Limit hosts
ansible-playbook playbook.yml --limit webservers
ansible-playbook playbook.yml --limit "web1,web2"

# Tags
ansible-playbook playbook.yml --tags "install,configure"
ansible-playbook playbook.yml --skip-tags "deploy"

# Variables
ansible-playbook playbook.yml -e "version=1.2.3"
ansible-playbook playbook.yml -e "@vars.yml"

# Check mode (dry run)
ansible-playbook playbook.yml --check

# Diff mode
ansible-playbook playbook.yml --diff

# Step through tasks
ansible-playbook playbook.yml --step

# Start at specific task
ansible-playbook playbook.yml --start-at-task="Install nginx"

# List tasks/hosts/tags
ansible-playbook playbook.yml --list-tasks
ansible-playbook playbook.yml --list-hosts
ansible-playbook playbook.yml --list-tags

# Verbose output
ansible-playbook playbook.yml -v    # verbose
ansible-playbook playbook.yml -vv   # more verbose
ansible-playbook playbook.yml -vvv  # debug
ansible-playbook playbook.yml -vvvv # connection debug
```

Chapter 9

Tasks

Tasks are the building blocks of Ansible automation. Each task calls a module to perform a specific action.

Task Keywords

Tasks support many options for controlling execution:

```
- name: Task description
  module_name:
    param1: value1
    param2: value2
  become: yes                      # Privilege escalation
  become_user: postgres            # User to become
  when: condition                  # Conditional execution
  loop: "{{ list_var }}"           # Iteration
  register: result                 # Store result
  ignore_errors: yes               # Continue on failure
  changed_when: false              # Override changed status
  failed_when: result.rc != 0     # Custom failure condition
  notify: Handler name             # Trigger handler
  tags: [install, configure]       # Tags for filtering
  delegate_to: localhost           # Run on different host
  run_once: true                   # Run only once
  retries: 3                       # Retry count
  delay: 5                         # Delay between retries
  until: result.rc == 0            # Retry until condition
  async: 300                       # Async timeout
  poll: 5                          # Async poll interval
  no_log: true                     # Hide sensitive output
  environment:                     # Environment variables
    PATH: "/opt/bin:{{ ansible_env.PATH }}"
```

Blocks

Blocks group tasks and provide error handling:

```
- name: Handle errors gracefully
  block:
    - name: Risky task
      command: /usr/bin/risky-command

    - name: Another risky task
      command: /usr/bin/another-risky

  rescue:
    - name: Recovery task
      debug:
        msg: "Something failed, recovering..."

  always:
    - name: Always runs
      debug:
        msg: "This always runs"
```

The `block` section contains tasks to attempt, `rescue` runs if any task fails, and `always` runs regardless of success or failure.

Chapter 10

Conditionals

Conditionals allow you to execute tasks based on variables, facts, or the results of previous tasks.

Basic Conditions

```
# Simple condition
- name: Install on Debian
  apt:
    name: nginx
  when: ansible_os_family == "Debian"

# Multiple conditions (AND)
- name: Install on Ubuntu 22.04
  apt:
    name: nginx
  when:
    - ansible_distribution == "Ubuntu"
    - ansible_distribution_version == "22.04"

# OR condition
- name: Install on Debian or Ubuntu
  apt:
    name: nginx
  when: ansible_distribution == "Debian" or ansible_distribution == "Ubuntu"
```

Common Conditional Patterns

```
# Variable defined
- name: Run if var defined
  debug:
    msg: "{{ my_var }}"
  when: my_var is defined

# Boolean
- name: Run if enabled
  command: /usr/bin/something
  when: feature_enabled | bool

# In list
- name: Install on specific distros
  apt:
    name: nginx
  when: ansible_distribution in ['Debian', 'Ubuntu']

# Result of previous task
- name: Check file
  stat:
    path: /etc/config
  register: config_file

- name: Create if missing
  file:
    path: /etc/config
    state: touch
  when: not config_file.stat.exists

# Combining with registered results
- name: Run command
  command: cat /etc/passwd
  register: passwd_contents
  changed_when: false

- name: Do something if root exists
  debug:
    msg: "Root user exists"
  when: "'root' in passwd_contents.stdout"
```

Chapter 11

Loops

Loops allow you to repeat a task multiple times, once for each item in a list. Instead of writing the same task three times for three packages, you write it once and loop over the packages.

How Loops Work

When you add `loop:` to a task, Ansible executes that task repeatedly—once for each item in the list. During each iteration, the current item is available as the special variable `{{ item }}`.

```
# WITHOUT a loop - repetitive and hard to maintain
- name: Install nginx
  apt:
    name: nginx
    state: present

- name: Install postgresql
  apt:
    name: postgresql
    state: present

- name: Install redis
  apt:
    name: redis
    state: present

# WITH a loop - clean and maintainable
- name: Install packages
  apt:
    name: "{{ item }}"
    state: present
  loop:
    - nginx
    - postgresql
    - redis
```

The looped version runs the `apt` task three times:

1. First iteration: `{{ item }}` = `nginx`
2. Second iteration: `{{ item }}` = `postgresql`
3. Third iteration: `{{ item }}` = `redis`

Basic Loops

```
# Loop over an inline list
- name: Install packages
  apt:
    name: "{{ item }}"
    state: present
  loop:
    - nginx
    - postgresql
    - redis

# Loop over a variable containing a list
- name: Install packages
  apt:
    name: "{{ item }}"
    state: present
  loop: "{{ packages }}"
  vars:
    packages:
      - nginx
      - postgresql
      - redis
```

```
# Loop to create multiple directories
- name: Create application directories
  file:
    path: "{{ item }}"
    state: directory
    mode: '0755'
  loop:
    - /opt/app
    - /opt/app/config
    - /opt/app/logs
    - /var/run/app
```

Looping Over Dictionaries

When looping over a list of dictionaries, access each field using `item.fieldname`:

```
# Each item is a dictionary with 'name' and 'groups' keys
- name: Create users
  user:
    name: "{{ item.name }}"
    groups: "{{ item.groups }}"
    shell: "{{ item.shell | default('/bin/bash') }}"
    state: present
  loop:
    - { name: 'alice', groups: 'admin', shell: '/bin/zsh' }
    - { name: 'bob', groups: 'developers' }
    - { name: 'charlie', groups: 'developers' }

# More readable multi-line format
- name: Create users
  user:
    name: "{{ item.name }}"
    groups: "{{ item.groups }}"
  loop:
    - name: alice
      groups: admin
    - name: bob
      groups: developers
```

Looping Over a Dictionary Variable

Use the `dict2items` filter to loop over a dictionary's key-value pairs:

```
vars:
  users:
    alice:
      uid: 1001
      home: /home/alice
    bob:
      uid: 1002
      home: /home/bob

tasks:
  # item.key = 'alice', item.value = {uid: 1001, home: /home/alice}
  - name: Create users from dictionary
    user:
      name: "{{ item.key }}"
      uid: "{{ item.value.uid }}"
      home: "{{ item.value.home }}"
    loop: "{{ users | dict2items }}"
```

Nested Loops

Use the `product` filter to loop over combinations of two lists:

```
# Creates: /home/alice/docs, /home/alice/downloads,
#          /home/bob/docs, /home/bob/downloads
- name: Create user directories
  file:
    path: "/home/{{ item.0 }}/{{ item.1 }}"
    state: directory
    owner: "{{ item.0 }}"
```

```
  loop: "{{ users | product(dirs) | list }}"
  vars:
    users: ['alice', 'bob']
    dirs: ['docs', 'downloads']
```

Loop Control

Fine-tune loop behavior with `loop_control`:

```
# Get the current index (0-based by default)
- name: Show position in loop
  debug:
    msg: "Item {{ index }}: {{ item }}"
  loop:
    - apple
    - banana
    - cherry
  loop_control:
    index_var: index

# Customize the output label (useful for large dictionaries)
- name: Create users
  user:
    name: "{{ item.name }}"
    uid: "{{ item.uid }}"
    comment: "{{ item.comment }}"
  loop: "{{ user_list }}"
  loop_control:
    label: "{{ item.name }}"  # Only shows name, not entire dict

# Pause between iterations (useful for rate-limited APIs)
- name: Call API for each user
  uri:
    url: "https://api.example.com/users/{{ item }}"
    method: POST
  loop: "{{ users }}"
  loop_control:
    pause: 2  # Wait 2 seconds between each call
```

Registering Loop Output

When you register a loop's output, you get a list of results:

```
- name: Check if files exist
  stat:
    path: "{{ item }}"
  loop:
    - /etc/hosts
    - /etc/passwd
    - /etc/shadow
  register: file_stats

# file_stats.results is a list of stat results
- name: Show missing files
  debug:
    msg: "{{ item.item }} does not exist"
  loop: "{{ file_stats.results }}"
  when: not item.stat.exists
```

Running Multiple Tasks Per Item

A `loop:` only applies to a single task—you cannot directly share one loop across multiple tasks. However, there are several patterns to achieve similar results:

Pattern 1: Use `include_tasks` with a loop

Create a separate file with the tasks you want to run for each item, then include it with a loop:

```
# main.yml
- name: Configure each application
```

```
  include_tasks: configure_app.yml
  loop:
    - { name: 'web', port: 80 }
    - { name: 'api', port: 8080 }
    - { name: 'admin', port: 9000 }

# configure_app.yml (runs once per item)
- name: Create directory for {{ item.name }}
  file:
    path: "/opt/{{ item.name }}"
    state: directory

- name: Deploy config for {{ item.name }}
  template:
    src: app.conf.j2
    dest: "/opt/{{ item.name }}/config.conf"

- name: Start {{ item.name }} service
  service:
    name: "{{ item.name }}"
    state: started
```

Pattern 2: Define the list once and repeat it

Store the list in a variable and reference it in each task:

```
vars:
  apps:
    - { name: 'web', port: 80 }
    - { name: 'api', port: 8080 }
    - { name: 'admin', port: 9000 }

tasks:
  - name: Create app directories
    file:
      path: "/opt/{{ item.name }}"
      state: directory
    loop: "{{ apps }}"

  - name: Deploy configs
    template:
      src: app.conf.j2
      dest: "/opt/{{ item.name }}/config.conf"
    loop: "{{ apps }}"

  - name: Start services
    service:
      name: "{{ item.name }}"
      state: started
    loop: "{{ apps }}"
```

Pattern 3: Use a role with loop

Roles receive the loop variable and can run multiple tasks with it:

```
# playbook.yml
- name: Configure apps
  include_role:
    name: app_setup
  vars:
    app_name: "{{ item.name }}"
    app_port: "{{ item.port }}"
  loop:
    - { name: 'web', port: 80 }
    - { name: 'api', port: 8080 }
```

The key difference between the patterns:

- **Pattern 1 (include_tasks)**: Tasks run in sequence *per item* (all tasks for item 1, then all tasks for item 2)

- **Pattern 2 (repeated loop)**: Each task completes for *all items* before the next task starts (create all directories, then deploy all configs, then start all services)

Choose based on whether you need all tasks for one item to complete before moving to the next item.

Legacy Syntax

These older `with_*` constructs still work but `loop` is preferred:

```
with_items: "{{ list }}"        # Use: loop: "{{ list }}"
with_dict: "{{ dict }}"         # Use: loop: "{{ dict | dict2items }}"
with_file: "{{ files }}"        # Use: loop + lookup('file', item)
with_fileglob: "*.txt"          # Use: loop + fileglob lookup
with_sequence: start=1 end=10   # Use: loop: "{{ range(1, 11) }}"
```

Chapter 12

Variables

Variables make your playbooks flexible and reusable. Ansible has a sophisticated variable precedence system.

Variable Precedence

Variables are loaded from multiple sources. Higher numbers override lower ones:

1. Role defaults (`roles/x/defaults/main.yml`)
2. Inventory file or script group vars
3. Inventory `group_vars/all`
4. Playbook `group_vars/all`
5. Inventory `group_vars/*`
6. Playbook `group_vars/*`
7. Inventory file or script host vars
8. Inventory `host_vars/*`
9. Playbook `host_vars/*`
10. Host facts / cached set_facts
11. Play vars
12. Play `vars_prompt`
13. Play `vars_files`
14. Role vars (`roles/x/vars/main.yml`)
15. Block vars
16. Task vars
17. `include_vars`
18. `set_facts` / registered vars
19. Role parameters
20. `include` parameters
21. Extra vars (`-e` command line) **always win**

Defining Variables

```
# In playbook
vars:
  http_port: 80
  max_clients: 200

# From file
vars_files:
  - vars/main.yml
  - "vars/{{ ansible_os_family }}.yml"

# Prompt user
vars_prompt:
  - name: username
    prompt: "Enter username"
    private: no
  - name: password
    prompt: "Enter password"
    private: yes
    encrypt: sha512_crypt

# Register task output
- command: hostname
  register: hostname_result
```

```
# Set fact
- set_fact:
    my_var: "{{ some_value }}"
    cacheable: yes
```

Accessing Variables

```
# Simple
"{{ variable }}"

# Dict
"{{ user.name }}"
"{{ user['name'] }}"

# List
"{{ users[0] }}"
"{{ users[0].name }}"

# Default value
"{{ variable | default('default_value') }}"

# Omit parameter if undefined
parameter: "{{ variable | default(omit) }}"

# Combining
"{{ list_var | join(',') }}"
"{{ dict_var | to_json }}"
```

Special Variables

Ansible provides many built-in variables:

Variable	Description
`inventory_hostname`	Name in inventory
`ansible_hostname`	Discovered short hostname
`ansible_fqdn`	Discovered FQDN
`ansible_host`	Address to connect to
`groups`	Dict of all groups and hosts
`group_names`	List of groups current host belongs to
`hostvars`	Dict of all hosts' variables
`ansible_facts`	Facts gathered about current host
`ansible_play_hosts`	Hosts in current play
`ansible_play_batch`	Current batch of hosts
`inventory_dir`	Inventory directory path
`playbook_dir`	Playbook directory path
`role_path`	Current role path

Chapter 13

Facts

Facts are system information automatically gathered from managed nodes. They include details about the operating system, network interfaces, hardware, and more.

Controlling Fact Gathering

```
# Disable fact gathering
- hosts: all
  gather_facts: no

# Explicit gathering
- name: Gather facts
  setup:

# Filtered gathering
- setup:
    gather_subset:
      - network
      - virtual
    filter: ansible_eth*
```

Custom Facts

Create custom facts by placing executable scripts or INI files in `/etc/ansible/facts.d/`:

```
# File: /etc/ansible/facts.d/custom.fact
[general]
app_version=1.0
```

Access custom facts via:

```
"{{ ansible_local.custom.general.app_version }}"
```

Common Facts

Fact	Description
`ansible_distribution`	Ubuntu, CentOS, etc.
`ansible_distribution_version`	22.04, 8, etc.
`ansible_distribution_release`	jammy, focal, etc.
`ansible_os_family`	Debian, RedHat, etc.
`ansible_architecture`	x86_64
`ansible_kernel`	5.15.0-generic
`ansible_hostname`	Short hostname
`ansible_fqdn`	Full hostname
`ansible_default_ipv4.address`	Primary IP
`ansible_all_ipv4_addresses`	All IPv4 addresses

`ansible_memtotal_mb`	Total RAM
`ansible_processor_vcpus`	CPU count
`ansible_env`	Environment variables
`ansible_pkg_mgr`	apt, yum, dnf, etc.
`ansible_service_mgr`	systemd, sysvinit, etc.

Chapter 14

Templates (Jinja2)

Templates let you generate dynamic configuration files using Jinja2 syntax. Variables, loops, and conditionals can all be used within templates.

Basic Template

```
{# templates/nginx.conf.j2 #}
# Managed by Ansible - do not edit
server {
    listen {{ http_port }};
    server_name {{ server_name }};

    location / {
        root {{ document_root }};
    }
}
```

Using Templates

```
- name: Deploy nginx config
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/sites-available/default
    owner: root
    group: root
    mode: '0644'
    backup: yes
    validate: nginx -t -c %s
  notify: Restart nginx
```

Jinja2 Syntax

```
{# Comments #}

{# Variables #}
{{ variable }}
{{ user.name }}
{{ items[0] }}

{# Conditionals #}
{% if condition %}
  ...
{% elif other_condition %}
  ...
{% else %}
  ...
{% endif %}

{# Loops #}
{% for item in items %}
  {{ item }}
{% endfor %}

{% for key, value in dict.items() %}
  {{ key }}: {{ value }}
{% endfor %}

{# Loop variables #}
{% for item in items %}
  {{ loop.index }}      {# 1-indexed #}
  {{ loop.index0 }}     {# 0-indexed #}
  {{ loop.first }}      {# True if first #}
  {{ loop.last }}       {# True if last #}
  {{ loop.length }}     {# Total items #}
{% endfor %}

{# Whitespace control #}
```

```
{%- if condition -%}   {# Strip whitespace #}
  ...
{%- endif -%}

{# Include other templates #}
{% include 'header.j2' %}

{# Macros #}
{% macro input(name, value='', type='text') %}
  <input type="{{ type }}" name="{{ name }}" value="{{ value }}">
{% endmacro %}

{{ input('username') }}
```

Common Filters

Filters transform data in templates and playbooks:

```
{# String operations #}
{{ name | upper }}
{{ name | lower }}
{{ name | capitalize }}
{{ text | trim }}
{{ text | replace('old', 'new') }}
{{ items | join(', ') }}

{# Numbers #}
{{ value | int }}
{{ value | float }}
{{ value | round(2) }}

{# Lists #}
{{ list | first }}
{{ list | last }}
{{ list | length }}
{{ list | sort }}
{{ list | unique }}

{# Data formats #}
{{ data | to_json }}
{{ data | to_yaml }}
{{ json_string | from_json }}

{# Defaults #}
{{ var | default('fallback') }}
{{ var | default(omit) }}

{# Path manipulation #}
{{ path | basename }}
{{ path | dirname }}

{# Hashing #}
{{ string | hash('sha256') }}
{{ string | password_hash('sha512') }}
{{ string | b64encode }}
```

Chapter 15

Handlers

Handlers are tasks that only run when notified by other tasks. They're typically used to restart services after configuration changes.

```
handlers:
  - name: Restart nginx
    service:
      name: nginx
      state: restarted
    listen: "restart web services"

  - name: Reload nginx
    service:
      name: nginx
      state: reloaded

tasks:
  - name: Update config
    template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    notify:
      - Restart nginx

  - name: Another task
    copy:
      src: site.conf
      dest: /etc/nginx/sites-available/
    notify: "restart web services"

  # Force handler execution immediately
  - name: Force handlers now
    meta: flush_handlers
```

Key Points:

- Handlers run at the end of a play, after all tasks
- Each handler runs only once, even if notified multiple times
- Use `listen` to trigger multiple handlers with one notification
- Use `meta: flush_handlers` to run handlers immediately

Chapter 16

Roles

Roles provide a structured way to organize related tasks, variables, files, and templates into reusable components.

Role Structure

```
roles/webserver/
+-- tasks/
|   +-- main.yml         # Task entry point
+-- handlers/
|   +-- main.yml         # Handler definitions
+-- templates/
|   +-- nginx.conf.j2    # Jinja2 templates
+-- files/
|   +-- index.html       # Static files
+-- vars/
|   +-- main.yml         # High-priority variables
+-- defaults/
|   +-- main.yml         # Default variables (lowest priority)
+-- meta/
|   +-- main.yml         # Role dependencies
+-- README.md
```

Using Roles

```
# Classic syntax
- hosts: webservers
  roles:
    - common
    - webserver
    - { role: app, tags: ['app'] }
    - role: database
      vars:
        db_name: myapp
      when: inventory_hostname in groups['dbservers']

# Include syntax (dynamic)
- hosts: webservers
  tasks:
    - name: Apply common role
      include_role:
        name: common
      tags: common

# Import syntax (static)
- hosts: webservers
  tasks:
    - import_role:
        name: webserver
      vars:
        http_port: 8080
```

Role Dependencies

Define role dependencies in `meta/main.yml`:

```
---
dependencies:
  - role: common
  - role: nginx
    vars:
      http_port: 8080
  - role: geerlingguy.docker
```

Creating and Installing Roles

```
# Create role skeleton
ansible-galaxy role init my_role
```

```
# Install from Galaxy
ansible-galaxy role install geerlingguy.docker

# Install from requirements.yml
ansible-galaxy install -r requirements.yml

# requirements.yml
roles:
  - name: geerlingguy.docker
  - name: my_role
    src: https://github.com/user/repo
    version: v1.0.0

collections:
  - name: community.general
    version: ">=5.0.0"
```

Chapter 17

Collections

Collections are the distribution format for Ansible content, including roles, modules, and plugins.

```
# Install collection
ansible-galaxy collection install community.general

# Install from requirements.yml
ansible-galaxy collection install -r requirements.yml

# List installed
ansible-galaxy collection list
```

Using Collections

```
# FQCN (Fully Qualified Collection Name)
- name: Use collection module
  community.general.ufw:
    rule: allow
    port: 22

# Or with collections keyword
- hosts: all
  collections:
    - community.general
  tasks:
    - name: Use module
      ufw:
        rule: allow
        port: 22
```

Chapter 18

Vault

Ansible Vault encrypts sensitive data like passwords, keys, and certificates.

Vault Commands

```
# Create encrypted file
ansible-vault create secrets.yml

# Edit encrypted file
ansible-vault edit secrets.yml

# Encrypt existing file
ansible-vault encrypt vars.yml

# Decrypt file
ansible-vault decrypt vars.yml

# Change password
ansible-vault rekey secrets.yml

# View encrypted file
ansible-vault view secrets.yml

# Encrypt string
ansible-vault encrypt_string 'secret_value' --name 'variable_name'
```

Using Vault

```
# Run with vault password prompt
ansible-playbook playbook.yml --ask-vault-pass

# Use password file
ansible-playbook playbook.yml --vault-password-file ~/.vault_pass

# Multiple vault IDs
ansible-playbook playbook.yml \
  --vault-id dev@~/.vault_dev \
  --vault-id prod@~/.vault_prod

# Inline encrypted variable
db_password: !vault |
  $ANSIBLE_VAULT;1.1;AES256
  61626364656667686970...

# Reference vault file
vars_files:
  - vars/main.yml
  - vars/vault.yml
```

Chapter 19

Common Modules

This section covers the most frequently used Ansible modules.

Package Management

```
# apt (Debian/Ubuntu)
- apt:
    name: "{{ packages }}"
    state: present
    update_cache: yes
    cache_valid_time: 3600
  vars:
    packages:
      - nginx
      - postgresql

# yum/dnf (RHEL/CentOS/Fedora)
- dnf:
    name: nginx
    state: latest

# pip
- pip:
    name: flask
    version: "2.0.0"
    virtualenv: /opt/venv

# Generic package module
- package:
    name: nginx
    state: present
```

Files and Directories

```
# file - manage files/directories
- file:
    path: /etc/app
    state: directory
    owner: app
    group: app
    mode: '0755'
    recurse: yes

# copy - copy files
- copy:
    src: files/config.conf
    dest: /etc/app/config.conf
    owner: root
    group: root
    mode: '0644'
    backup: yes

# lineinfile - manage lines in files
- lineinfile:
    path: /etc/hosts
    line: "192.168.1.100 app.local"
    state: present

# blockinfile - manage blocks in files
- blockinfile:
    path: /etc/hosts
    block: |
      192.168.1.100 app1.local
      192.168.1.101 app2.local
    marker: "# {mark} ANSIBLE MANAGED BLOCK"
```

Services

```
# service
- service:
    name: nginx
    state: started
    enabled: yes

# systemd
- systemd:
    name: nginx
    state: restarted
    daemon_reload: yes
    enabled: yes
```

Users and Groups

```
# user
- user:
    name: deploy
    groups: sudo,docker
    append: yes
    shell: /bin/bash
    generate_ssh_key: yes
    ssh_key_bits: 4096
    password: "{{ password | password_hash('sha512') }}"

# group
- group:
    name: developers
    state: present

# authorized_key
- authorized_key:
    user: deploy
    key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
    state: present
```

Commands

```
# command - simple commands (no shell features)
- command: /usr/bin/app --config /etc/app.conf
  args:
    chdir: /opt/app
    creates: /var/run/app.pid

# shell - shell commands (pipes, redirects work)
- shell: cat /etc/passwd | grep root > /tmp/root.txt
  args:
    executable: /bin/bash

# script - run local script on remote
- script: scripts/setup.sh
  args:
    creates: /var/run/setup.done
```

Networking

```
# uri - HTTP requests
- uri:
    url: https://api.example.com/health
    method: GET
    return_content: yes
    status_code: 200
  register: api_health

# get_url - download files
- get_url:
    url: https://example.com/app.tar.gz
    dest: /tmp/app.tar.gz
    checksum: sha256:abc123...

# wait_for - wait for conditions
- wait_for:
    port: 8080
```

```
    host: 127.0.0.1
    delay: 5
    timeout: 300
```

Debug and Testing

```
# debug
- debug:
    msg: "Variable value: {{ my_var }}"

- debug:
    var: hostvars[inventory_hostname]

# assert
- assert:
    that:
      - ansible_distribution == "Ubuntu"
      - ansible_distribution_version is version('20.04', '>=')
    fail_msg: "Requires Ubuntu 20.04 or later"

# pause
- pause:
    prompt: "Press enter to continue"
```

Chapter 20

Includes and Imports

Ansible provides two mechanisms for including external content: static imports and dynamic includes.

```
# import_tasks - static (processed at parse time)
- import_tasks: tasks/setup.yml
  vars:
    package: nginx

# include_tasks - dynamic (processed at runtime)
- include_tasks: "tasks/{{ ansible_os_family }}.yml"

# import_playbook
- import_playbook: webservers.yml

# include_vars
- include_vars: "vars/{{ ansible_os_family }}.yml"
```

Key Differences

Feature	`import_*`	`include_*`
Processing	Parse time (static)	Runtime (dynamic)
Variables in path	No	Yes
Conditionals	Applied to each task	Applied to include
Tags	Inherited by tasks	Not inherited
Loops	Not supported	Supported

Chapter 21

Error Handling

Ansible provides several mechanisms for handling errors gracefully.

```
# Ignore errors
- command: /may/fail
  ignore_errors: yes

# Ignore unreachable hosts
- command: /usr/bin/app
  ignore_unreachable: yes

# Custom failure condition
- command: /usr/bin/app
  register: result
  failed_when: "'ERROR' in result.stdout"

# Custom changed condition
- command: /usr/bin/check
  register: result
  changed_when: result.rc == 2

# Retry until success
- uri:
    url: http://localhost:8080/health
  register: result
  until: result.status == 200
  retries: 10
  delay: 5

# Stop all hosts on any error
- hosts: all
  any_errors_fatal: true
  tasks: ...

# Maximum failure percentage
- hosts: all
  max_fail_percentage: 25
  tasks: ...
```

Chapter 22

Delegation and Local Actions

Delegation allows tasks to run on a *different* host than the current play target. This is useful for orchestration tasks like updating load balancers, sending notifications, or interacting with APIs from a central location.

Note: Delegation is *not* for jump/bastion host access. For that, see the Bastion Hosts section below.

Delegating Tasks

```
# Delegate to another host - runs on lb.example.com but uses
# variables from the current host (inventory_hostname)
- name: Add server to load balancer
  command: /usr/bin/lb add {{ inventory_hostname }}
  delegate_to: lb.example.com

# Remove from load balancer before maintenance
- name: Remove from load balancer
  uri:
    url: "http://lb.example.com/api/pools/web/members/{{ ansible_host }}"
    method: DELETE
  delegate_to: localhost
```

Local Actions

Run tasks on the Ansible control node:

```
# Wait for host to become reachable (run from control node)
- name: Wait for SSH
  wait_for:
    host: "{{ ansible_host }}"
    port: 22
    timeout: 300
  delegate_to: localhost

# Equivalent using local_action
- name: Wait for SSH
  local_action:
    module: wait_for
    host: "{{ ansible_host }}"
    port: 22

# Make API call from control node
- name: Notify deployment tracker
  uri:
    url: https://deploy.example.com/api/deployments
    method: POST
    body_format: json
    body:
      server: "{{ inventory_hostname }}"
      version: "{{ app_version }}"
  delegate_to: localhost
```

Run Once

Execute a task only once for the entire play, regardless of how many hosts:

```
# Database migration - only run once, not per host
- name: Run database migrations
  command: /opt/app/migrate.sh
  run_once: true
  delegate_to: "{{ groups['dbservers'][0] }}"

# Send single Slack notification for deployment
- name: Notify Slack
  slack:
    token: "{{ slack_token }}"
    msg: "Deployed {{ app_version }} to {{ ansible_play_hosts | length }} servers"
```

```
  run_once: true
  delegate_to: localhost
```

Bastion/Jump Hosts

For accessing hosts through a bastion (jump) host, configure SSH proxying—this is separate from delegation. The bastion host acts as an intermediary for SSH connections to private hosts.

Method 1: ansible.cfg configuration

```
# ansible.cfg
[ssh_connection]
ssh_args = -o ProxyJump=bastion.example.com
# Or for older SSH versions:
# ssh_args = -o ProxyCommand="ssh -W %h:%p bastion.example.com"
```

Method 2: Inventory variables

```
# inventory
[private_servers]
app1.internal ansible_host=10.0.1.10
app2.internal ansible_host=10.0.1.11
db1.internal ansible_host=10.0.1.20

[private_servers:vars]
ansible_ssh_common_args='-o ProxyJump=ubuntu@bastion.example.com'
```

Method 3: SSH config file

```
# ~/.ssh/config
Host bastion
    HostName bastion.example.com
    User ubuntu
    IdentityFile ~/.ssh/bastion_key

Host 10.0.1.*
    User deploy
    ProxyJump bastion
    IdentityFile ~/.ssh/internal_key
```

Then Ansible connects automatically through the bastion:

```
# Playbook - no special configuration needed
- hosts: private_servers
  tasks:
    - name: This runs on private hosts via bastion
      apt:
        name: nginx
        state: present
```

Method 4: Group-specific bastion

```
# group_vars/private_servers.yml
ansible_ssh_common_args: >-
  -o ProxyJump=ubuntu@bastion.example.com:22
  -o StrictHostKeyChecking=no
```

Dynamic bastion selection:

```
# inventory
[bastions]
bastion-eu.example.com region=eu
bastion-us.example.com region=us

[eu_servers]
eu-app1.internal bastion=bastion-eu.example.com

[us_servers]
us-app1.internal bastion=bastion-us.example.com
```

```
[eu_servers:vars]
ansible_ssh_common_args='-o ProxyJump={{ bastion }}'

[us_servers:vars]
ansible_ssh_common_args='-o ProxyJump={{ bastion }}'
```

Chapter 23

Async Tasks

For long-running operations, async tasks allow Ansible to continue without waiting.

```
# Fire and forget (poll: 0)
- name: Long running task
  command: /usr/bin/long_task
  async: 3600
  poll: 0
  register: long_task

# Check status later
- name: Check task status
  async_status:
    jid: "{{ long_task.ansible_job_id }}"
  register: job_result
  until: job_result.finished
  retries: 100
  delay: 10
```

Chapter 24

Lookups

Lookups retrieve data from external sources during playbook execution. They run on the control node and return data that can be used in tasks, templates, and variables.

File Lookup

Read the contents of a file on the control node:

```
# Read file content into a variable
- name: Deploy SSH public key
  authorized_key:
    user: deploy
    key: "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"

# Use file content in a template variable
- name: Set message of the day
  copy:
    content: "{{ lookup('file', 'files/motd.txt') }}"
    dest: /etc/motd
```

Environment Variable Lookup

Access environment variables from the control node:

```
# Use environment variable in task
- name: Clone repository
  git:
    repo: "https://{{ lookup('env', 'GITHUB_TOKEN') }}@github.com/org/repo.git"
    dest: /opt/app

# Set variable from environment
vars:
  home_dir: "{{ lookup('env', 'HOME') }}"
  deploy_env: "{{ lookup('env', 'DEPLOY_ENVIRONMENT') | default('development') }}"
```

Password Lookup

Generate or retrieve passwords (creates file if it doesn't exist):

```
# Generate and store a database password
- name: Create database user
  mysql_user:
    name: appuser
    password: "{{ lookup('password', 'credentials/db_password length=20') }}"
    priv: 'myapp.*:ALL'

# Generate password with specific characters
- name: Set admin password
  user:
    name: admin
    password: "{{ lookup('password', '/dev/null length=16 chars=ascii_letters,digits') | password_hash('sha512') }}"
```

Pipe Lookup

Run a command on the control node and capture output:

```
# Get current date for backup naming
- name: Create backup
  archive:
    path: /var/www
    dest: "/backups/www-{{ lookup('pipe', 'date +%Y%m%d-%H%M%S') }}.tar.gz"

# Get git commit hash
- name: Tag deployment
  copy:
```

```
    content: "Deployed: {{ lookup('pipe', 'git rev-parse HEAD') }}"
    dest: /opt/app/VERSION
```

URL Lookup

Fetch content from a URL:

```
# Fetch JSON configuration from API
- name: Configure application
  copy:
    content: "{{ lookup('url', 'https://config.example.com/app.json') }}"
    dest: /etc/app/config.json

# Parse JSON from URL
- name: Get latest release version
  set_fact:
    latest_version: "{{ (lookup('url', 'https://api.github.com/repos/org/app/releases/latest') | from_json).tag_name }}"
```

Template Lookup

Render a Jinja2 template and return the result:

```
# Render template to variable
- name: Generate config from template
  set_fact:
    app_config: "{{ lookup('template', 'templates/config.yml.j2') }}"

# Use rendered template as content
- name: Deploy inline config
  copy:
    content: "{{ lookup('template', 'nginx-site.conf.j2') }}"
    dest: /etc/nginx/sites-available/mysite
```

First Found Lookup

Return the first file that exists from a list:

```
# Load environment-specific variables
- name: Include environment config
  include_vars: "{{ lookup('first_found', params) }}"
  vars:
    params:
      files:
        - "{{ ansible_distribution }}-{{ ansible_distribution_version }}.yml"
        - "{{ ansible_distribution }}.yml"
        - "{{ ansible_os_family }}.yml"
        - default.yml
      paths:
        - vars

# Copy first available template
- name: Deploy config
  template:
    src: "{{ lookup('first_found', ['templates/{{ env }}.conf.j2', 'templates/default.conf.j2']) }}"
    dest: /etc/app/app.conf
```

Combining Lookups

Lookups can be combined with filters for powerful data manipulation:

```
# Read file and parse as YAML
- set_fact:
    config: "{{ lookup('file', 'config.yml') | from_yaml }}"

# Read lines from file into a list
- set_fact:
    allowed_hosts: "{{ lookup('file', 'allowed_hosts.txt').splitlines() }}"

# Lookup with error handling
- set_fact:
    api_key: "{{ lookup('env', 'API_KEY') | default(lookup('file', 'api_key.txt'), true) }}"
```

Chapter 25

Filters Reference

Filters transform data in Jinja2 expressions. Here's a comprehensive reference:

String Filters

```
{{ name | upper }}                    # UPPERCASE
{{ name | lower }}                    # lowercase
{{ name | capitalize }}               # Capitalize
{{ name | title }}                    # Title Case
{{ text | trim }}                     # Strip whitespace
{{ text | regex_replace('a', 'b') }}  # Regex replace
{{ text | quote }}                    # Shell quote
```

Collection Filters

```
{{ list | first }}                    # First element
{{ list | last }}                     # Last element
{{ list | length }}                   # Count
{{ list | sort }}                     # Sort
{{ list | unique }}                   # Unique values
{{ list | flatten }}                  # Flatten nested lists
{{ list | map('upper') | list }}      # Map filter
{{ list | select('match', 'a.*') }}   # Select matching
{{ list1 | union(list2) }}            # Union
{{ list1 | intersect(list2) }}        # Intersection
{{ list1 | difference(list2) }}       # Difference
```

Math Filters

```
{{ value | int }}                     # To integer
{{ value | float }}                   # To float
{{ value | round(2) }}                # Round
{{ list | sum }}                      # Sum
{{ list | max }}                      # Maximum
{{ list | min }}                      # Minimum
```

Data Structure Filters

```
{{ data | to_json }}                  # To JSON
{{ data | to_nice_json(indent=2) }}   # Pretty JSON
{{ data | to_yaml }}                  # To YAML
{{ json_str | from_json }}            # Parse JSON
{{ dict | dict2items }}               # Dict to list
{{ list | items2dict }}               # List to dict
{{ dict | combine(other) }}           # Merge dicts
```

Path Filters

```
{{ path | basename }}                 # File name
{{ path | dirname }}                  # Directory
{{ path | expanduser }}               # Expand ~
{{ path | realpath }}                 # Resolve symlinks
```

Cryptographic Filters

```
{{ string | hash('sha256') }}         # Hash
{{ password | password_hash('sha512') }} # Password hash
{{ data | b64encode }}                # Base64 encode
{{ encoded | b64decode }}             # Base64 decode
```

Chapter 26

Testing and Debugging

Validating and troubleshooting playbooks is essential for reliable automation.

Command-Line Validation

```
# Syntax check
ansible-playbook playbook.yml --syntax-check

# List tasks
ansible-playbook playbook.yml --list-tasks

# List hosts
ansible-playbook playbook.yml --list-hosts

# Check mode (dry run)
ansible-playbook playbook.yml --check

# Diff mode (show changes)
ansible-playbook playbook.yml --diff --check

# Step through
ansible-playbook playbook.yml --step

# Start at task
ansible-playbook playbook.yml --start-at-task="task name"

# Debug with verbosity
ansible-playbook playbook.yml -vvvv

# Test inventory
ansible-inventory -i inventory --list
ansible-inventory -i inventory --graph
```

In-Playbook Debugging

```
# Print variable
- debug:
    var: my_variable

# Print message
- debug:
    msg: "Value is {{ my_variable }}"

# Conditional debug
- debug:
    var: result
  when: result is failed

# Pause for inspection
- pause:
    prompt: "Check the output above"

# Assert conditions
- assert:
    that:
      - my_var is defined
      - my_var | length > 0
    fail_msg: "my_var is not valid"
```

Chapter 27

Performance Tips

Optimize Ansible execution for faster deployments.

Configuration Optimizations

```
# ansible.cfg
[defaults]
forks = 20                          # Parallel processes
gathering = smart                    # Cache facts
fact_caching = jsonfile
fact_caching_connection = /tmp/facts
fact_caching_timeout = 86400

[ssh_connection]
pipelining = True                    # Reduce SSH operations
ssh_args = -o ControlMaster=auto -o ControlPersist=60s
```

Playbook Optimizations

```
# Disable fact gathering when not needed
- hosts: all
  gather_facts: no

# Selective fact gathering
- hosts: all
  gather_facts: yes
  gather_subset:
    - network
    - hardware

# Free strategy (parallel task execution)
- hosts: all
  strategy: free

# Async for long tasks
- command: /long/running/task
  async: 3600
  poll: 0

# Batch hosts (rolling updates)
- hosts: all
  serial: 5           # 5 at a time
  serial: "20%"       # 20% at a time
  serial: [1, 5, 10]  # Rolling: 1, then 5, then 10

# Limit package operations
- apt:
    name: "{{ packages }}"  # Install all at once
    state: present
```

Chapter 28

Best Practices

Follow these guidelines for maintainable, reliable Ansible automation:

1. **Use roles** for reusable content
2. **Keep secrets in Vault** — never commit plaintext passwords
3. **Use meaningful names** for tasks and variables
4. **Test with `--check`** before running
5. **Use version control** for playbooks and inventory
6. **Tag tasks** for selective execution
7. **Use `ansible-lint`** for code quality
8. **Document roles** with README files
9. **Use `group_vars` and `host_vars`** instead of hardcoding
10. **Prefer `state: present`** over `state: latest` for stability
11. **Use handlers** for service restarts
12. **Keep playbooks idempotent** — safe to run multiple times
13. **Use FQCN** for modules (e.g., `ansible.builtin.apt`)
14. **Pin collection versions** in requirements.yml
15. **Use CI/CD** to test playbooks automatically

Chapter 29

Quick Command Reference

Essential commands for everyday Ansible use:

```
# Inventory
ansible-inventory --list
ansible-inventory --graph

# Ad-hoc
ansible all -m ping
ansible all -m setup
ansible all -a "uptime"

# Playbook
ansible-playbook site.yml
ansible-playbook site.yml -i inventory
ansible-playbook site.yml --limit host1
ansible-playbook site.yml --tags deploy
ansible-playbook site.yml --check --diff
ansible-playbook site.yml -e "var=value"

# Vault
ansible-vault create secrets.yml
ansible-vault edit secrets.yml
ansible-vault encrypt_string 'secret'

# Galaxy
ansible-galaxy role init myrole
ansible-galaxy install -r requirements.yml
ansible-galaxy collection install community.general

# Documentation
ansible-doc apt
ansible-doc -l          # List all modules
ansible-doc -t lookup file
```

Version: Ansible 2.15+ / Python 3.9+

Published by URADICAL