DevOps Shack

ANSIBLE DECLARATIVE CONFIGURATION

BY DEVOPS SHACK

Click here for DevSecOps & Cloud DevOps Course

DevOps Shack

Ansible Declarative Configuration – Step-by-Step Guide

Table of Contents

- o Creating and using roles for reusability

- o Role directory structure (tasks/, handlers/, templates/, etc.)

- o Role-based playbook example

- o Using Ansible Galaxy for public roles

6. **Running, Debugging & Best Practices**

- o Running playbooks with ansible-playbook

- o Dry-run (--check), verbosity (-v)

- o Debugging tasks and using ansible-lint

- o Recommended folder structure and Git versioning

Ç Step 1: Introduction & Installation

◆ What is

Ansible is an open-source automation tool used for **configuration management**, **application deployment**, **orchestration**, and **task automation**. It allows IT administrators and DevOps engineers to define the state of systems in a simple, **declarative way using human-readable YAML files**, known as **playbooks**.

Unlike other tools (like Puppet or Chef), Ansible is **agentless**—it only requires SSH access to target machines and Python installed on them. This simplicity makes Ansible lightweight, easy to adopt, and ideal for both small teams and large-scale infrastructure.

◆ Declarative vs. Imperative Configuration

Ansible embraces the **declarative** approach. This means you describe *what* state you want the system to be in (e.g., "nginx should be installed and running"), and Ansible figures out *how* to make it so.

In contrast, **imperative** tools require you to explicitly define every step in sequence (e.g., install nginx, enable nginx, start nginx).

Declarative configurations are:

- Easier to maintain

- More predictable

- Naturally idempotent (no unintended side effects if you run the same playbook multiple times)

◆ Installing Ansible

Let's get started by installing Ansible on your local machine (Control Node).

Requirements:

- Python 3.8+

- pip (Python package manager)

- Internet connection

- Basic understanding of the command line

Installation on Ubuntu/Debian:

sudo apt update

sudo apt install software-properties-common

sudo add-apt-repository --yes --update ppa:ansible/ansible sudo

apt install ansible

Installation on macOS (using Homebrew):

brew install ansible

Installation on Windows (via WSL or Python/pip):

For Windows, it's best to use **WSL (Windows Subsystem for Linux)**:

1. Enable WSL and install Ubuntu from the Microsoft Store

2. Open the Ubuntu terminal and follow the Ubuntu steps

above Or install with pip:

pip install ansible

💡 Tip: If you plan to use Ansible heavily, prefer WSL2 or a Linux VM for best compatibility.

◆ **Verifying the Installation**

Once Ansible is installed, verify it by checking the version:

ansible --version

You should see output like: ansible

[core 2.16.1]

config file = /etc/ansible/ansible.cfg

python version = 3.10.x

If you see that, you're all set!

◆ Optional: Set Up a Python Virtual Environment

To isolate your Ansible environment:

python3 -m venv ansible-env

source ansible-env/bin/activate

pip install ansible

This is a good practice when working on multiple projects.

🟩 Summary

In this step, you've:

- Learned what Ansible is and how it works

- Understood the difference between declarative and imperative configuration

- Installed Ansible on your system

- Verified that it works

With Ansible installed, you're ready to start creating your first Ansible project, define an inventory, and write your first playbook!

Ç Step 2: Project Setup & Inventory

Now that Ansible is installed, it's time to set up your first Ansible project and create an **inventory**, which is a key part of how Ansible knows what machines to manage.

◆   Understanding Ansible Architecture

Before we jump into the hands-on part, let's quickly understand the core concepts of Ansible's architecture:

- **Control Node**: The machine from which Ansible commands are run (your local machine).

- **Managed Nodes** (or hosts/targets): The remote machines you want to configure.

- **Inventory File**: A file that lists your managed nodes.

- **Playbook**: A YAML file that defines the desired configuration (more on this in the next step).

- **Modules**: The units of work in Ansible (e.g., install package, copy files, start service).

Ansible communicates with managed nodes via SSH (port 22), so make sure passwordless SSH is set up.

◆   Setting Up Your First Ansible Project

You can keep your Ansible projects well-organized by using a dedicated folder for each one. Here's how you can start:

mkdir my-ansible-project

cd my-ansible-project

Inside this directory, you'll store:

- inventory.ini: your list of managed nodes

- playbook.yml: your Ansible instructions

- group_vars/ or host_vars/: variable definitions

- roles/: reusable role directories

◆ **Writing the Inventory File**

Create a file named inventory.ini in the root of your project:

[webservers]

192.168.1.101

192.168.1.102

[dbservers]

db1.example.com ansible_user=ubuntu ansible_port=22

Explanation:

- You define **groups** of servers using [groupname].

- You can assign variables like ansible_user, ansible_port, etc., on a per- host basis.

You can also use a **YAML-style inventory** (inventory.yml) if you prefer:

```
all:
 children:
  webservers:
   hosts:
    192.168.1.101:
    192.168.1.102:
  dbservers:
   hosts:
     db1.example.com:
      ansible_user: ubuntu
      ansible_port: 22
```

'' Choose either .ini or .yml format — both work, but .ini is more beginners.

◆ **SSH Access to Remote Machines**

Make sure your control node (your local machine) can SSH into the target machines without needing a password each time.

Generate SSH Key (if not already done):

ssh-keygen -t rsa -b 4096

Copy Key to Remote Server:

ssh-copy-id ubuntu@192.168.1.101

Repeat for each server.

◆ **Test Your Setup**

You can now test Ansible's ability to connect to the nodes in your inventory:

ansible all -i inventory.ini -m ping

Expected output:

192.168.1.101 | SUCCESS => {

  "changed": false,

  "ping": "pong"

}

If you get a **"UNREACHABLE"** message, check:

- IP address

- SSH key setup

- ansible_user value

- Whether the machine is up and reachable

◆ **Using ansible.cfg (Optional)**

You can simplify command-line usage by creating an ansible.cfg file in project root:

[defaults]

inventory = inventory.ini

host_key_checking = False

retry_files_enabled = False

This lets you run commands like:

ansible all -m ping

...without needing the -i inventory.ini every time.

Summary

By completing this step, you have:

- Understood how Ansible's architecture works

- Created a clean folder structure for your project

- Written your first inventory file (INI or YAML)

- Set up passwordless SSH between your control node and managed nodes

- Verified your setup using the Ansible ping module

Ç Step 3: Writing Declarative Playbooks

Now that you have your Ansible project and inventory set up, it's time to write your first **playbook** — the core of Ansible's declarative automation.

◆ **What is an Ansible Playbook?**

An **Ansible Playbook** is a YAML file that defines a set of tasks to run on managed nodes. Each playbook describes:

- **What** systems to configure (via inventory or group)

- **How** they should be configured (tasks, modules, variables)

- **What state** you want

(declaratively) Playbooks are:

- **Human-readable**

- **Idempotent** (safe to run multiple times without unintended side effects)

- Reusable and modular

◆ **Basic YAML Syntax (Quick Primer)**

Ansible playbooks are written in **YAML**. Here's a quick guide:

- Use spaces, **not tabs**

- Indentation is crucial

- Use : after keys, and prefix lists with

- Example:

- name: My First Playbook

  hosts: webservers

become: yes

  tasks:

```
- name: Install Nginx

  apt:

    name: nginx

    state: present
```

◆ Anatomy of a Playbook

Let's break down each part:

```
---

- name: Configure Web Server   # Play name

  hosts: webservers         # Target group from inventory

  become: yes             # Use sudo (for privilege escalation)


  tasks:              # List of tasks to execute
    - name: Ensure Nginx is installed

      apt:

        name: nginx

        state: present

        update_cache: yes


    - name: Ensure Nginx is running

      service:

        name: nginx

        state: started

        enabled: true
```

Explanation:

- name: A descriptive label for the play or task

- hosts: Group or host from your inventory

- become: Use sudo to perform privileged tasks

- tasks: A list of operations to apply

- Modules used: apt, service — this is where the **declarative logic** happens

◆ **Using Common Ansible Modules Declaratively**

Here are some frequently used modules:

Module	Purpose	Example (declarative state)
apt / yum	Install packages	state: present or absent
file	Manage files/directories	state: directory, mode: 0755
copy	Copy local files to remote	src, dest, mode
service	Control services	state: started, enabled: true
user	Manage users	state: present, shell: /bin/bash

All these modules work **declaratively** — you just define the end state.

◆ **Running Your Playbook**

Once your playbook is ready, run it using the ansible-playbook command:

ansible-playbook -i inventory.ini playbook.yml

With ansible.cfg set up (as shown in Step 2), you can simplify:

ansible-playbook playbook.yml

Expected output:

PLAY [Configure Web Server] *** TASK

[Ensure Nginx is installed] ***

changed: [192.168.1.101]

TASK [Ensure Nginx is running] ***

13

ok: [192.168.1.101]

■ ok means the task was already in the correct state

changed means the task made modifications to reach the desired state

+ failed means something went wrong

◆ **Dry Run (Check Mode)**

You can test a playbook without making changes:

ansible-playbook playbook.yml --check

Useful for validation before deploying to production.

■ **Summary**

In this step, you've:

- Learned how Ansible playbooks work

- Understood YAML syntax and playbook structure

- Used common modules declaratively

- Created and executed your first real playbook

- Verified and validated your configuration

Ansible now ensures your systems reach (and stay in) the exact state you want — with no manual work or surprises.

Ç Step 4: Variables, Templates & Handlers

As your Ansible projects grow, hardcoding values into your playbooks becomes inefficient and error-prone. This step introduces **variables**, **templates**, and **handlers** — essential tools for writing dynamic and clean configurations.

◆ **Using Variables in Ansible**

Variables allow you to customize your playbooks without rewriting them for different environments or hosts.

◆ **Types of Variables:**

1. **Inline Variables (inside**

playbooks): vars:

  app_port: 8080

2. **Group Variables** (for an entire group): Create a folder named group_vars/ and file like group_vars/webservers.yml:

app_port: 8080

3. **Host Variables** (for a specific host): Create a folder named host_vars/ and file like host_vars/192.168.1.101.yml:

nginx_enabled: true

4. **Extra Variables (CLI-based)**:

ansible-playbook site.yml -e "app_port=8080"

◆ **Using Variables in Tasks**

Once a variable is defined, you can use it like this:

```
- name: Use variable in task
  copy:
    src: app.conf
    dest: /etc/myapp/{{ app_port }}.conf
```

Ansible uses **Jinja2 templating** to evaluate variables — which brings us to the next topic…

◆ **Templating with Jinja2**

Templates allow you to define configuration files (e.g., nginx.conf) dynamically, with variable substitution.

◆ **Example:**

Create a template file templates/nginx.conf.j2:

```
server {

    listen {{ app_port }};

    server_name {{ server_name }};

    root /var/www/{{ site_folder }};

}
```

In your playbook:

```
- name: Deploy Nginx config from template

  template:

    src: templates/nginx.conf.j2

    dest: /etc/nginx/sites-available/default
```

◆ **Variables file:**

```
server_name: example.com

site_folder: html
```

This allows the same template to work across different environments.

I. Make sure the destination path has correct permissions or use become: yes.

◆ **Handlers (Reactive Tasks)**

Handlers are special tasks that run only when notified by another task. They're used for tasks that should only happen if something changes — like restarting a service after a config update.

◆ **Example:**

tasks:

  - name: Deploy nginx config

    template:

      src: templates/nginx.conf.j2

      dest: /etc/nginx/sites-available/default

    notify: Restart Nginx

handlers:

  - name: Restart Nginx

    service:

      name: nginx

      state: restarted

If the file changes, Ansible will trigger the Restart Nginx handler. If not, it skips it.

This approach keeps your playbooks efficient and avoids unnecessary restarts.

◆ **Putting It All Together**

Here's a full playbook example that combines everything:

- name: Setup web server with variables and templates

  hosts: webservers

  become: yes

  vars:

```yaml
  app_port: 8080

  server_name: mysite.com

  site_folder: html


tasks:
 - name: Install Nginx

   apt:

     name: nginx

     state: present

     update_cache: yes


 - name: Deploy Nginx config

   template:

     src: templates/nginx.conf.j2

     dest: /etc/nginx/sites-available/default

   notify: Restart Nginx


handlers:
 - name: Restart Nginx

   service:

     name: nginx

     state: restarted
```

🟩 Summary

In this step, you've learned how to:

- Use variables to make your playbooks dynamic

18

- Create Jinja2 templates for reusable configs

- Define handlers to respond to changes

- Build more modular and maintainable infrastructure

Ç Step 5: Organizing with Roles

As your Ansible playbooks grow in complexity, it becomes necessary to break things down into **roles** — reusable, organized units that can be shared across projects and teams.

Roles help you maintain structure, promote reuse, and reduce duplication.

◆ What is an Ansible Role?

A **role** is a collection of related files and directories (tasks, variables, templates, handlers, etc.) that define a reusable automation component.

You can create roles for:

- Installing a web server

- Setting up a database
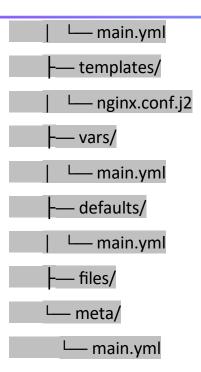
- Deploying an application

- Configuring a system service

Roles can be reused in multiple playbooks or shared publicly via **Ansible Galaxy**.

◆ Role Directory Structure

Here's the standard layout of a role: css

CopyEdit

```
my-ansible-project/
└── roles/
    └── nginx/
        ├── tasks/
        │   └── main.yml
        ├── handlers/
```

```
|   └── main.yml
├── templates/
|   └── nginx.conf.j2
├── vars/
|   └── main.yml
├── defaults/
|   └── main.yml
├── files/
└── meta/
    └── main.yml
```

Each folder serves a purpose:

- tasks/: Core tasks for the role

- handlers/: Notifications

- templates/: Jinja2 template files

- vars/: Variables with higher precedence

- defaults/: Default variables

- files/: Static files to copy

- meta/: Metadata about the role (dependencies, author, etc.)

◆ **Creating a Role**

Use Ansible's built-in generator:

ansible-galaxy init roles/nginx

This scaffolds a new role named nginx inside the roles/ folder.

◆ **Writing Role Content**

Let's say you want to create a role for installing and configuring **Nginx**.

Example: roles/nginx/tasks/main.yml

```yaml
---
- name: Install nginx
  apt:
    name: nginx
    state: present
    update_cache: yes


- name: Deploy nginx config
  template:
    src: nginx.conf.j2
    dest: /etc/nginx/sites-available/default
  notify: Restart nginx
```

Example: roles/nginx/handlers/main.yml

yaml

CopyEdit

```yaml
---
- name: Restart nginx
  service:
    name: nginx
    state: restarted
```

Example: roles/nginx/templates/nginx.conf.j2

```
server {
    listen {{ app_port }};
    server_name {{ server_name }};
    root /var/www/{{ site_folder }};
```

}

Example: roles/nginx/defaults/main.yml

app_port: 80

server_name: localhost

site_folder: html

◆ **Using a Role in a Playbook**

After creating your role, reference it in your main playbook like this:

- name: Deploy web server using nginx role

  hosts: webservers

  become: yes

  roles:

    - nginx

That's it! Ansible will automatically look for the nginx role in roles/nginx/ and run all tasks, handlers, and templates.

◆ **Role Variables: Flexibility and Override**

You can override role defaults by passing variables:

- name: Use nginx role with custom vars

  hosts: webservers

  become: yes

  roles:

```
- role: nginx

  vars:

    app_port: 8080

    server_name: myapp.local
```

Or use host/group vars as shown earlier.

◆ Sharing Roles via Ansible Galaxy (Optional)

If you want to share your role publicly, you can:

1. Create a GitHub repo for your role

2. Tag and push it

3. Publish it on

galaxy.ansible.com Or install other

people's roles:

ansible-galaxy install geerlingguy.nginx

🟩 Summary

In this step, you've:

- Learned what Ansible roles are and why they matter

- Understood the folder structure and responsibilities

- Created your own nginx role with templates and handlers

- Used roles in your playbooks for better organization

- Explored how to customize and reuse them easily

Roles turn your one-off playbooks into **production-grade infrastructure automation**.

Ç Step 6: Testing & Best Practices

You've built a strong foundation: writing playbooks, managing variables, using templates and handlers, and modularizing everything with roles. Now it's time to **test**, **validate**, and **follow best practices** to ensure your infrastructure is reliable, secure, and maintainable.

◆ Why Testing Matters in Ansible

Ansible configurations **change infrastructure**, and without testing, a small error can:

- Break your app

- Misconfigure services

- Bring down production

Testing reduces human error and ensures consistent deployments.

◆ Types of Testing in Ansible

1. Syntax Checks

Before running a playbook, always do a syntax

check: ansible-playbook playbook.yml --syntax-

check

This catches indentation issues, unknown modules, malformed YAML, etc.

2. Dry Run (Check Mode)

Simulate changes without applying them:

ansible-playbook playbook.yml --check

It tells you what *would* change, which is great for reviewing changes before pushing to production.

3. Linting (Static Code Analysis)

Use ansible-lint to catch common anti-patterns:

ansible-lint playbook.yml

You'll get warnings for:

- Deprecated modules

- Hardcoded values

- Missing handlers

- Improper role

structures Install with:

pip install ansible-lint

4. Molecule for Role Testing

Molecule is a tool designed for **testing Ansible roles** in isolated environments (Docker, Vagrant, etc.).

To install:

pip install molecule[docker]

To test a role:

cd roles/nginx

molecule init scenario -r nginx -d docker

molecule test

Molecule runs:

- Linting

- Dependency install

- Converge (provisioning)

- Verify (testing)

- Destroy

Great for **CI/CD pipelines** and local test cycles.

◆ Best Practices for Writing Ansible

■ 1. Use Roles from Day One

Even for small projects, roles promote clean structure and reusability. Avoid long, monolithic playbooks.

■ 2. Stick to Declarative Patterns

Use state: present/absent, enabled: true, and similar idempotent options. Avoid scripting logic unless necessary.

■ 3. Use Variables and Defaults

Keep playbooks flexible by extracting values into defaults/, vars/, or external files.

■ 4. Document Everything

- Use name: for each task (makes output readable)
- Add inline comments for templates and tricky vars
- Version control everything with Git

■ 5. Limit Hardcoding

Avoid hardcoding values in tasks. Use variables, group/host vars, or encrypted secrets with **Ansible Vault**.

◆ Optional: Ansible Vault for Secrets

To keep sensitive info (passwords, API keys) safe:

Create a vault file:

ansible-vault create secrets.yml

Edit an existing vault file:

ansible-vault edit

secrets.yml **Use in**

playbooks:

vars_files:

  - secrets.yml

Run playbooks with vault:

ansible-playbook playbook.yml --ask-vault-pass

Or use a vault password file for automation.

◆ **Integrating with CI/CD**

Ansible can be run in pipelines like GitHub Actions, GitLab CI, Jenkins, etc.

A sample GitHub Action might:

- Checkout repo

- Set up Python and Ansible

- Run linting and playbooks

- Use Molecule for testing roles

Automation ensures every change is tested before deployment.

Summary

In this final step, you've learned to:

- Test playbooks safely using syntax check, dry run, and linting

- Use Molecule to test roles like code

- Protect secrets using Ansible Vault

- Structure your code for readability and reusability

- Integrate Ansible into DevOps workflows using CI/CD

— ✕ ◯   Conclusion

Ansible empowers you to automate infrastructure with clarity, consistency, and control. Through this 6-step guide, you've gone from the fundamentals to advanced best practices—transforming your configurations into well-structured, declarative, and reusable automation code.

You started by setting up Ansible and understanding how inventory and playbooks define your infrastructure. Then, you explored the power of
variables, templates, and handlers to keep things dynamic and DRY (Don't Repeat Yourself). From there, you modularized your approach using roles, laying the groundwork for scalable, team-friendly automation. Finally, you
strengthened your practice with testing strategies, secrets management, and battle-tested best practices to make your playbooks production-grade.

By following these steps, you've not only learned **how to write Ansible code**—you've learned how to write it **the right way**.

Now, whether you're managing a few servers or scaling across data centers and cloud environments, your infrastructure is under version control, reproducible, and secure. You can confidently integrate it into your CI/CD workflows and collaborate with teams across environments.

Next Steps & Recommendations:

- Explore **Ansible Galaxy** for ready-made roles

- Dive into **Ansible Tower / AWX** for GUI-based control

- Start integrating Ansible into **CI/CD pipelines**

- Keep your skills sharp with **real-world use cases**