BY FENIL GAJJAR

ANSIBLE DAILY TASKS

ANSIBLE PLAYBOOKS & MODULES

- COMPERHENSIVE GUIDE
- THEORY + PRACITCAL TASKS
- REAL TIME SCENARIO TASKS
- END TO END DOC

Contact Us:

fenilgajjar.devops@gmail.com
linkedin.com/in/fenil-gajjar
github.com/Fenil-Gajjar

🚀 Automate Like a Pro: The DevOps Guide to Ansible Playbooks & Modules 🛠️

👋 Welcome & Thank You!

First of all — **a huge thank you** to everyone who has followed and supported my previous documentation series 🙏
Your feedback, encouragement, and enthusiasm keep this journey alive. I truly appreciate the time you've spent reading, sharing, and learning from the content I've created.

Now, I'm excited to bring you another value-packed resource:

🚀 Automate Like a Pro: The DevOps Guide to Ansible Playbooks & Modules

This guide is more than just a walkthrough — it's a **complete hands-on journey** through the core of what makes Ansible powerful:
Playbooks and Modules — the very heart of infrastructure automation.

You'll learn not just what they are, but:

- 🧠 How they work together
- 🔍 How to structure them properly
- 🛠️ How to use them in *real-world tasks* DevOps engineers actually face
- ⚙️ Plus: best practices, troubleshooting tips, and reusable automation patterns

Whether you're just getting started or looking to sharpen your skills, this guide is designed to be **clear, practical, and production-minded** — exactly how modern DevOps documentation should be.

Let's get into it — and as always, thank you again for reading! 🙌

🗒️ Introduction to Ansible Playbooks

Ansible is a powerful automation tool used for configuration management, application deployment, and orchestration. At the heart of this automation lies the **Ansible Playbook**—a YAML-based file that defines **what** needs to be done and **how** to do it, in a clear and structured manner.

✅ What is a Playbook?

An **Ansible Playbook** is a blueprint of automation tasks. It describes a list of actions (called "plays") that need to be executed on one or more remote systems. Each play targets a group of hosts and defines a set of **tasks**, which are executed using **Ansible modules**.

Think of playbooks as human-readable instructions for machines—concise, repeatable, and scalable.

📌 Why Use Playbooks?

Here's why playbooks are essential in modern infrastructure management:

- **Consistency**: Same set of tasks applied across environments without manual intervention.

- **Idempotency**: Re-running playbooks doesn't break the system; it ensures the desired state.

- **Clarity**: Written in YAML, playbooks are easy to read, review, and version-control.

- **Declarative Style**: You describe the *end state*, and Ansible figures out *how* to get there.

- **Extensibility**: Easily integrate with roles, variables, templates, and more for larger projects.

🧠 How Playbooks Work

Playbooks work by:

1. Connecting to the defined hosts (via SSH or WinRM).

2. Running each **task** sequentially using appropriate **modules**.

3. Ensuring that each task changes the system only if needed (idempotent behavior).

4. Reporting results in a clean and informative way.

✍️ A Simple Example

```
- name: Install and start Apache web server
  hosts: webservers
  become: yes

  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present

    - name: Start Apache service
      service:
        name: apache2
        state: started
        enabled: yes
```

This playbook:

- Runs on hosts in the `webservers` group.
- Installs Apache using the `apt` module.
- Ensures the Apache service is running and enabled at boot using the `service` module.

🛠️ When to Use Playbooks

Playbooks are ideal when:

- You need to automate a **sequence** of tasks.
- You want to manage configuration **at scale**.
- You're implementing **CI/CD** pipelines or **infra-as-code**.
- You want a **version-controlled** approach to infrastructure.

🎯 Importance of Playbooks in Automation

Ansible playbooks are the **core engine** behind infrastructure automation. While ad-hoc commands are suitable for one-time tasks, playbooks provide the **repeatability**, **structure**, and **reliability** that enterprise-grade automation demands.

🔄 Repeatability & Consistency

Playbooks ensure that automation tasks execute the same way every time, across all environments—development, staging, and production. This eliminates "configuration drift" and manual inconsistencies.

> **Example**: Installing NGINX on 10 servers with the same config? One playbook does it uniformly, every time.

⚙️ Declarative Infrastructure as Code (IaC)

Playbooks allow you to define **desired state**, not the step-by-step instructions to get there. This abstracts complexity and aligns with modern DevOps practices like IaC.

> **You declare**: “NGINX must be installed and running.”
> **Ansible ensures**: That’s the final result, regardless of the server's current state.

👥 Scalability Across Environments

With inventory grouping and variables, playbooks can target:

- A single host
- A specific group of servers
- An entire data center or cloud region

This makes it easy to scale operations from local VMs to thousands of machines.

📚 Documentation & Auditing

Playbooks double as **living documentation** of your infrastructure. Anyone on your team can read and understand:

- What configurations are applied
- When changes were made (via version control)
- Why certain tasks exist (with descriptions/comments)

🛡️ Safe, Idempotent Execution

Playbooks can be safely re-run multiple times without causing adverse effects. Ansible checks whether changes are needed before taking action, preventing disruptions or duplication.

> Re-running a playbook won't reinstall a package if it's already there—it only makes changes when necessary.

🔌 Integration with DevOps Toolchains

Playbooks can be easily integrated into:

- **CI/CD pipelines** (e.g., Jenkins, GitLab CI)
- **Cloud provisioning tools** (e.g., Terraform)
- **Monitoring systems** (e.g., via Ansible callbacks)

This makes automation part of the **entire software lifecycle**, not just system setup.

🚀 Fast Rollouts & Disaster Recovery

Need to update 100 servers in minutes? Or reconfigure systems after an outage? Playbooks make it fast, safe, and auditable to push changes or recover systems.

🔧 What Are Ansible Modules?

Ansible modules are the **core building blocks** of automation in Ansible. Every task you define in a playbook uses a module under the hood to perform a specific action—like installing packages, managing users, copying files, or restarting services.

You can think of a module as a **small, standalone piece of code** that knows how to do *one thing* well.

🧱 Key Characteristics of Ansible Modules

- **Task-oriented**: Each module is designed for a particular job (e.g., install software, create users, manage services).
- **Idempotent**: Modules are built to ensure that running the same task multiple times doesn't cause unintended changes.
- **Cross-platform**: Many modules work on Linux, Windows, and cloud platforms.
- **Reusable**: The same module can be used across multiple playbooks and environments.

📁 Types of Modules

Modules are categorized based on their purpose:

Category	Description	Examples
System	Manage OS-level components	`user`, `group`, `hostname`
Package Manager	Install/remove software packages	`apt`, `yum`, `dnf`, `pip`
File	Work with files and directories	`copy`, `file`, `template`
Service	Start/stop/manage services	`service`, `systemd`
Cloud	Manage cloud infrastructure	`ec2`, `azure_rm`, `gcp_compute_instance`
Networking	Configure network devices or settings	`ios_config`, `nmcli`, `firewalld`
Utility	Help with debugging, pauses, conditions	`debug`, `pause`, `assert`
Custom	User-created modules for custom use cases	Written in Python or any language

✍️ How Modules Are Used in a Playbook

Every **task** in a playbook calls a module. For example:

```
- name: Install NGINX
  apt:
    name: nginx
    state: present
```

Here:

- `apt` is the module.
- It installs the `nginx` package and ensures it's present.
- Ansible handles the logic behind checking the current state and acting accordingly.

⚙️ Behind the Scenes

When you run a playbook:

1. Ansible connects to the remote host.

2. It sends the module code (or uses the built-in one if local).
3. The module executes the task on the target system.
4. The output is returned in JSON format to Ansible for processing and display.

👨‍💻 Writing Custom Modules

If your organization has unique requirements, you can write your own modules using:

- Python (most common)
- Bash or any executable scripting language
- Return output in JSON format for Ansible to parse

- Modules are the **action layer** of Ansible.
- They abstract away complex operations into simple YAML instructions.
- Understanding which module to use—and how—is key to writing effective automation.

🔗 Relationship Between Playbooks and Modules

Understanding the relationship between **Ansible Playbooks** and **Modules** is fundamental to mastering Ansible automation. In short:

> **Playbooks describe *what* to do, and Modules define *how* it's done.**

🧾 Playbooks: The Blueprint

A **playbook** is a YAML-formatted file that outlines the desired state of your systems. It tells Ansible:

- Which **hosts** to target
- What **tasks** to run
- In what **order**
- With what **conditions**, **variables**, and **privileges**

But the playbook itself doesn't *do* the actual work—it delegates that to modules.

🔧 Modules: The Workers

A **module** is a small program that performs a specific task on the remote system:

- Installing a package

- Creating a user
- Modifying a config file
- Restarting a service

Modules are **invoked** by tasks inside the playbook.

🛠️ How They Work Together

Each **task** in a playbook uses a module. For example:

```
- name: Ensure nginx is installed
  apt:
    name: nginx
    state: present
```

In this example:

- The **playbook** defines a task: "Ensure nginx is installed".
- The **task** uses the `apt` **module** to perform the actual installation.

- Ansible executes the module code on the remote machine, and reports back success/failure.

🔄 Execution Flow Summary

1. **You write** a playbook with tasks.
2. **Each task** specifies a module and arguments.
3. Ansible connects to the remote host and **runs the module**.
4. The module **performs the action** and returns results to Ansible.
5. Ansible **logs and displays** the output to you.

📚 Analogy

Imagine a playbook as a **movie script** and modules as the **actors**:

- The script says what scenes need to happen (tasks).
- The actors (modules) perform the actions.
- The director (Ansible) ensures the performance is in order, repeats if needed, and gives feedback.

🤝 Why This Relationship Matters

- It allows **separation of logic and execution**.
- You can reuse modules across many tasks and playbooks.
- Modules handle complexity; you just declare the outcome in YAML.
- Custom modules extend this relationship for your organization's specific needs.

So, a playbook **orchestrates**, while modules **execute**. Together, they form the complete automation engine of Ansible.

📘 Understanding Ansible Playbooks: A Complete Guide

Ansible playbooks are the **heart** of automation in Ansible. They're how you tell Ansible *what you want your systems to look like*. In this section, we'll break it all down—from syntax to structure to real-world tips.

🗒️ YAML Syntax Basics (Start Here)

Playbooks are written in **YAML** (YAML Ain't Markup Language), a simple and human-readable data format. YAML uses **indentation** and **key-value pairs** to describe configurations.

✅ Basic YAML Rules

- Use **spaces**, not tabs. Indentation is crucial.
- Use `:` to separate keys and values.
- Lists begin with `-` (dash and space).

🔍 Example

```
- name: Simple YAML example
  tasks:
    - name: Install nginx
      apt:
```

```
    name: nginx
    state: present
```

💡 *Tip: If your playbook fails unexpectedly, check your spacing first—it's the #1 cause of syntax errors!*

🧠 Anatomy of a Playbook

A playbook is made of **plays**, and each play runs a sequence of **tasks** on a group of **hosts**.

🔹 **Basic Components of a Playbook**

Here's what makes up a typical playbook:

```
---
- name: Configure web server
  hosts: webservers
  become: yes  # Run with sudo privileges

  vars:
    http_port: 80

  tasks:
```

```
  - name: Install Apache
    apt:
      name: apache2
      state: present

  - name: Ensure Apache is running
    service:
      name: apache2
      state: started
      enabled: yes
```

🧩 **Breakdown of Key Sections:**

Section	Purpose
`name`	Human-readable name for the play/task
`hosts`	Target group(s) of machines to run the play
`become`	Escalate privileges (e.g., sudo)

`vars`	Define custom variables
`tasks`	List of actions to perform
`modules`	Each task uses a module like `apt`, `service`, etc.

🔄 Multi-Play Playbooks

Playbooks can contain **multiple plays** to target different groups of servers in a single run.

📚 Example

```
---
- name: Configure web servers
  hosts: webservers
  become: yes
  tasks:
    - name: Install nginx
      apt:
        name: nginx
```

```
        state: present


- name: Configure database servers
  hosts: dbservers
  become: yes
  tasks:
    - name: Install MySQL
      apt:
        name: mysql-server
        state: present
```

🧠 *Think of this as automating different roles (web, DB, etc.) in a single deployment plan.*

🔁 Idempotency in Ansible

One of Ansible's most powerful features is **idempotency**—a fancy word that means:

> 🗣️ "If you run the same playbook 100 times, your system will stay in the same state after the first run unless something actually changes."

✅ **Why It Matters**

- Avoids duplicate tasks (e.g., doesn’t reinstall a package if it's already present).
- Safe to re-run playbooks in CI/CD pipelines or on servers.
- Helps prevent configuration drift over time.

🧪 **Example**

```
- name: Create a user
  user:
    name: devops
    state: present
```

This will:

- Add the user if not present.
- Do nothing if the user already exists.
- Never create duplicate users.

💡 *Tip: Most core Ansible modules are designed to be idempotent. Custom scripts may not be.*

📎 Real-World Playbook Tips

- **Comment generously**. Future-you or your teammates will thank you.
- **Use variables** to avoid hardcoding values (`http_port`, `app_version`, etc.).
- **Split roles** when your playbooks get big. Use `roles/` to organize them.
- **Always test** your playbook with `--check` and `--diff` when possible.

🧠 Understanding Ansible Modules: A Complete Guide

Ansible modules are the **workhorses** of your playbooks. While the playbook defines *what* you want, modules define *how it gets done*. Whether you're installing software, copying files, managing services, or configuring cloud resources—**a module is involved in every task**.

🔍 What is a Module in Ansible?

A **module** in Ansible is a small, reusable unit of code used to perform a specific task on a remote system. Every **task** you write in a playbook uses one module.

> 📌 **Think of a module as a function** that Ansible calls to do something on your target machine.

🔧 **Example:**

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

In this task:

- `apt` is the module.
- `name` and `state` are arguments passed to that module.

Ansible sends the task to the target host, runs the module there, and gets back a result (success/failure/changes made).

🏗️ Core Modules vs. Custom Modules

✅ Core Modules

These are the **built-in**, officially supported modules that come with Ansible by default. They are well-tested and regularly maintained.

- Examples: `copy`, `file`, `apt`, `yum`, `service`, `user`, `command`

✏️ Custom Modules

You can create your own modules if:

- A built-in module doesn't meet your need.
- You have a highly specific task or integration.
- You want to wrap custom logic in a reusable form.

Custom modules can be written in:

- Python (recommended)
- Bash
- Any executable language that returns JSON

> 💡 **Tip**: Custom modules should follow the Ansible interface (stdin for input, JSON for output) to integrate smoothly.

📂 Categories of Commonly Used Modules

Modules are grouped by function. Here's a breakdown of the most important and frequently used types:

🗂️ 1. File & Directory Management

Module	Purpose
`file`	Set permissions, create/remove files and directories
`copy`	Copy files from control node to target
`template`	Use Jinja2 templates to create config files

`stat`	Check file attributes

⚙️ 2. Service Management

Module	Purpose
`service`	Manage services via SysV/upstart/systemd
`systemd`	Modern service management

📦 3. Package Management

Module	Platform	Purpose
`apt`	Debian/Ubuntu	Install, remove packages
`yum`	RHEL/CentOS	Install, remove packages
`dnf`	Fedora/RHEL	Modern Yum replacement
`pip`	Python	Install Python packages

👤 4. User and Group Management

Module	Purpose
`user`	Manage system users
`group`	Manage system groups

🌐 5. Networking & Firewalls

Module	Purpose
`ufw`	Manage Uncomplicated Firewall (UFW)
`firewalld`	Manage firewalld
`iptables`	Manage iptables rules

☁️ 6. Cloud and Infrastructure

Provider	Example Modules
AWS	`ec2`, `s3`, `elb`, `route53`
Azure	`azure_rm_*`

GCP	`gcp_compute_instance`, `gcp_storage`
Docker	`docker_container`, `docker_image`
Kubernetes	`k8s`, `helm`

🔧 7. Utilities and Helpers

Module	Purpose
`debug`	Print messages in playbooks
`set_fact`	Define variables dynamically
`pause`	Pause execution for input/timing
`assert`	Validate conditions and fail with message

🧬 How Modules Work Behind the Scenes

Understanding what happens internally helps you troubleshoot and optimize your playbooks.

🛠️ Step-by-Step Process:

1. **Task is defined** in the playbook, using a module.
2. **Ansible connects** to the target host (typically via SSH).
3. The module (Python script or executable) is **transferred to the host**.
4. The module is **executed on the host**, using the arguments you passed in YAML.
5. The module **performs the action**, checks for changes, and returns a **JSON response**.
6. Ansible **captures the result**, updates its state, and shows you output.

📦 Example JSON Output from a Module:

```
{
  "changed": true,
  "msg": "nginx installed",
  "invocation": {
    "module_args": {
```

```
      "name": "nginx",
      "state": "present"
    }
  }
}
```

🚀 Why Understanding Modules Matters

- Helps you **write better playbooks** (more efficient, more readable).
- Makes debugging easier when things go wrong.
- Enables you to **extend** Ansible with custom functionality.
- Builds a foundation for **advanced topics** like roles, collections, and plugins.

🚀 Real-World Practical Task #1: Provision a Web Server (Nginx) with Custom Configuration

🧭 Use Case:

You're tasked with setting up a **production-ready Nginx web server** on a group of Ubuntu hosts. The server must:

- Install Nginx
- Serve content from a specific directory
- Have a custom HTML landing page
- Ensure Nginx is enabled and running
- Make sure UFW allows HTTP traffic

🧰 Modules Covered:

- `apt`
- `copy`
- `template`
- `file`

- service
- ufw

📁 Folder Structure (Recommended)

```
webserver-setup/
├── files/
│   └── index.html
├── templates/
│   └── nginx.conf.j2
└── playbook.yml
```

🔨 playbook.yml

```
---
- name: Setup Nginx Web Server
  hosts: webservers
  become: yes

  vars:
```

```
    web_root: /var/www/html
    nginx_port: 80

  tasks:

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

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

    - name: Copy custom index.html
      copy:
        src: files/index.html
        dest: "{{ web_root }}/index.html"
```

```
    owner: www-data
    group: www-data
    mode: '0644'

- name: Deploy custom nginx config
  template:
    src: templates/nginx.conf.j2
    dest: /etc/nginx/sites-available/default
  notify: Reload Nginx

- name: Ensure Nginx is started and enabled
  service:
    name: nginx
    state: started
    enabled: true

- name: Allow HTTP through UFW
  ufw:
    rule: allow
```

```
        port: "{{ nginx_port }}"
        proto: tcp

  handlers:
    - name: Reload Nginx
      service:
        name: nginx
        state: reloaded
```

📄 **files/index.html**

```
<!DOCTYPE html>
<html>
<head>
    <title>Welcome to Ansible Web Server</title>
</head>
<body>
    <h1>It works! 🚀</h1>
</body>
```

```
</html>
```

📝 **templates/nginx.conf.j2**

```
server {
    listen {{ nginx_port }};
    server_name _;
    root {{ web_root }};

    index index.html;

    location / {
        try_files $uri $uri/ =404;
    }
}
```

🧠 Why This Task Matters

- **Package installation**: `apt` is a go-to module in real-world Ubuntu environments.
- **File and template management**: `copy` and `template` show how to manage real content and configs.
- **Service handling**: `service` ensures your services are up and running.
- **Firewall configuration**: `ufw` ensures your server is reachable securely.

🛠️ These are foundational skills in **CI/CD pipelines, PaaS infrastructure, and provisioning workflows**.

✅ Result:

After running this playbook:

- Your Nginx web server is installed.
- A custom landing page is live.
- A custom config is deployed.
- Firewall allows traffic.
- It's ready for production or containerization.

🚀 Real-World Practical Task #2: Create a New System User with SSH Access for CI/CD Agent

🧭 Objective:

Create a dedicated Linux user (`ci-agent`) for CI/CD purposes. This user should:

- Exist on all target servers
- Have a valid home directory
- Be given a public SSH key for authentication
- Have sudo access (optional, controlled by a variable)

🔧 Modules Used:

- `user`
- `authorized_key`
- `copy`
- `lineinfile` or `template` (for sudoers)
- `group`

- `stat` (optional validation)

📁 Folder Structure

```
ci-user-setup/
├── files/
│   └── id_rsa.pub
└── playbook.yml
```

🛠️ `playbook.yml`

```
---
- name: Create a CI/CD System User with SSH Access
  hosts: all
  become: yes

  vars:
    ci_user: ci-agent
    ci_home: "/home/{{ ci_user }}"
```

```
    ci_ssh_pubkey: "{{ lookup('file', 'files/id_rsa.pub')
}}"
    grant_sudo: true

  tasks:

    - name: Ensure '{{ ci_user }}' group exists
      group:
        name: "{{ ci_user }}"
        state: present

    - name: Create '{{ ci_user }}' user
      user:
        name: "{{ ci_user }}"
        comment: "CI/CD Agent User"
        shell: /bin/bash
        group: "{{ ci_user }}"
        home: "{{ ci_home }}"
        create_home: yes
        state: present
```

```
- name: Create .ssh directory
  file:
    path: "{{ ci_home }}/.ssh"
    state: directory
    owner: "{{ ci_user }}"
    group: "{{ ci_user }}"
    mode: '0700'

- name: Add public SSH key for '{{ ci_user }}'
  authorized_key:
    user: "{{ ci_user }}"
    key: "{{ ci_ssh_pubkey }}"
    state: present
    manage_dir: no  # We already created the .ssh dir

- name: Optionally grant sudo access
  lineinfile:
    path: /etc/sudoers.d/{{ ci_user }}
```

```
    create: yes
    state: present
    line: "{{ ci_user }} ALL=(ALL) NOPASSWD:ALL"
    mode: '0440'
  when: grant_sudo
```

✅ Result:

- You now have a clean, automated, secure way to provision a system user with SSH access.
- This user can be used for **CI/CD pipelines**, **remote scripts**, or **infrastructure automation**.
- With `grant_sudo`, you can dynamically choose whether the user has privileged access.

💬 Why This Task Is Practical

- **Used in every cloud deployment or pipeline bootstrapping** step.
- Gives you reusable logic for bootstrapping secure remote users.
- Teaches critical modules (`user`, `authorized_key`, `lineinfile`) used in production and auditing tasks.

🚀 Real-World Practical Task #3: Deploy and Run a Dockerized Web App with Ansible

🧭 Objective:

Use Ansible to install Docker (if not present), deploy a containerized web application, and ensure it auto-starts on system reboot. This could be a simple `nginx`, `httpd`, or a real microservice container.

🔧 Modules Covered:

- `apt` (install dependencies)
- `docker_container`
- `docker_image`
- `service` (for Docker daemon)
- `uri` (optional – health check)

> 🐳 This task introduces the **Docker modules**, which are heavily used in cloud-native and containerized CI/CD pipelines.

📁 Folder Structure

```
docker-deploy/
├── playbook.yml
└── files/
    └── daemon.json  # (Optional: Docker daemon config)
```

📖 Anatomy of This Playbook

Section	Purpose
`pre_tasks`	Ensure dependencies are present
`tasks`	Install Docker, pull image, run container
`vars`	Define Docker image, port, etc.
`handlers`	Restart Docker if config changes

🛠️ **playbook.yml**

```
---
- name: Deploy Dockerized Web Application
  hosts: app_servers
  become: yes

  vars:
    docker_package_state: present
    docker_service_state: started
    docker_image_name: nginx
    docker_image_tag: latest
    docker_container_name: webapp
    docker_container_port: 80
    host_port: 8080

  pre_tasks:

    - name: Install required packages
```

```
      apt:
        name:
          - apt-transport-https
          - ca-certificates
          - curl
          - gnupg
          - lsb-release
        state: present
        update_cache: yes

    - name: Add Docker GPG key
      apt_key:
        url: https://download.docker.com/linux/ubuntu/gpg
        state: present

    - name: Add Docker repository
      apt_repository:
        repo: "deb [arch=amd64]
https://download.docker.com/linux/ubuntu {{
ansible_distribution_release }} stable"
```

```
      state: present
    notify: Update APT cache

tasks:

  - name: Install Docker
    apt:
      name: docker-ce
      state: "{{ docker_package_state }}"
    notify: Restart Docker

  - name: Ensure Docker service is running
    service:
      name: docker
      state: "{{ docker_service_state }}"
      enabled: true

  - name: Pull Docker image "{{ docker_image_name }}"
    docker_image:
```

```
        name: "{{ docker_image_name }}"
        tag: "{{ docker_image_tag }}"
        source: pull

    - name: Ensure "{{ docker_container_name }}" container
is running
      docker_container:
        name: "{{ docker_container_name }}"
        image: "{{ docker_image_name }}:{{ docker_image_tag
}}"
        state: started
        restart_policy: always
        published_ports:
          - "{{ host_port }}:{{ docker_container_port }}"

    - name: (Optional) Health check for the web app
      uri:
        url: "http://localhost:{{ host_port }}"
        return_content: yes
        status_code: 200
```

```
    register: web_health
    retries: 3
    delay: 5
    until: web_health.status == 200

handlers:

  - name: Restart Docker
    service:
      name: docker
      state: restarted

  - name: Update APT cache
    apt:
      update_cache: yes
```

✅ What This Playbook Achieves

- Installs and configures Docker on any Ubuntu host

- Deploys a Dockerized `nginx` container exposed on port 8080
- Ensures it auto-restarts on reboot or crash
- Optionally performs a live health check on the container

🔥 Why This Task Is DevOps Gold

- Recreates an end-to-end delivery of an app in a **cloud-native way**
- Embraces **infrastructure-as-code**, containerization, and **self-healing**
- Builds foundational knowledge to integrate with **CI/CD, Kubernetes, ECS, EKS, and more**

🚀 Real-World Practical Task #4: Configure and Manage a Load Balancer using HAProxy

🧭 Objective:

Set up **HAProxy** on a designated load balancer server to distribute traffic to a backend pool of web servers. This setup is essential in high-availability, scalable infrastructure environments.

🛠️ Modules Covered:

- `apt` – Install HAProxy
- `template` – Deploy HAProxy config dynamically
- `service` – Manage HAProxy
- `lineinfile` – For optional sysctl/network tuning
- `copy` – Optional: deploy SSL certs or error pages

📖 **Use Case Summary:**

- One **HAProxy node** acts as a load balancer
- Backend web servers (e.g., nginx) are assumed to be up and running
- Load balancer listens on port 80
- Balances HTTP traffic across multiple web servers

📁 **Folder Structure**

```
haproxy-setup/
├── playbook.yml
└── templates/
    └── haproxy.cfg.j2
```

🛠️ playbook.yml

```
---
- name: Configure HAProxy Load Balancer
  hosts: loadbalancer
```

```
  become: yes


  vars:
    haproxy_cfg_path: /etc/haproxy/haproxy.cfg
    frontend_port: 80
    backend_servers:
      - name: web1
        address: 192.168.56.101
        port: 80
      - name: web2
        address: 192.168.56.102
        port: 80


  tasks:


    - name: Install HAProxy
      apt:
        name: haproxy
        state: present
```

```
      update_cache: yes

  - name: Deploy HAProxy configuration
    template:
      src: templates/haproxy.cfg.j2
      dest: "{{ haproxy_cfg_path }}"
      mode: '0644'
    notify: Restart HAProxy

  - name: Ensure HAProxy service is running and enabled
    service:
      name: haproxy
      state: started
      enabled: true

handlers:
  - name: Restart HAProxy
    service:
      name: haproxy
```

```
        state: restarted
```

📄 templates/haproxy.cfg.j2

```
global
    log /dev/log local0
    log /dev/log local1 notice
    daemon
    maxconn 256

defaults
    log     global
    mode    http
    option  httplog
    option  dontlognull
    retries 3
    timeout connect 5000ms
    timeout client  50000ms
```

```
    timeout server  50000ms


frontend http_front
    bind *:{{ frontend_port }}
    default_backend http_back


backend http_back
{% for srv in backend_servers %}
    server {{ srv.name }} {{ srv.address }}:{{ srv.port }}
check
{% endfor %}
```

✅ **Outcome:**

- A ready-to-use HAProxy load balancer
- Distributes incoming HTTP traffic between backend nodes
- Automatically reconfigurable via template updates
- Ensures HAProxy starts on boot and restarts when the config changes

💡 **Real-Life Application:**

- Used in **production for high availability and failover**
- Common in microservices and monolith-to-microservice transition environments
- Great foundation for introducing **TLS termination**, **health checks**, or **blue-green deployments**

🔐 Real-World Practical Task #5: Automated User Management with Ansible

🧭 Objective:

Use Ansible to:

- Create multiple users across multiple servers
- Add SSH public keys for secure access
- Assign (or revoke) sudo privileges
- Remove unwanted users safely

This is one of the most **frequently automated tasks** in infrastructure management, especially in **team onboarding/offboarding** or **multi-node fleet configuration**.

🛠️ Modules Covered:

- `user`
- `authorized_key`
- `group`

- lineinfile
- copy

📁 Folder Structure

```
user-management/
├── playbook.yml
├── group_vars/
│   └── all.yml
└── files/
    └── john.pub
```

📄 group_vars/all.yml

```
users_to_create:
  - username: john
    shell: /bin/bash
    ssh_key: "{{ lookup('file', 'files/john.pub') }}"
    sudo: true
```

```
  - username: alice
    shell: /bin/zsh
    ssh_key: "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQD..."  # paste her key directly
    sudo: false

users_to_remove:
  - mark
  - tempuser
```

🛠️ playbook.yml

```
---
- name: Manage system users
  hosts: all
  become: yes

  tasks:

    - name: "Create user '{{ item.username }}'"
```

```
    user:
      name: "{{ item.username }}"
      shell: "{{ item.shell | default('/bin/bash') }}"
      state: present
      create_home: yes
    loop: "{{ users_to_create }}"
    tags: [create, users]

  - name: "Add SSH key for '{{ item.username }}'"
    authorized_key:
      user: "{{ item.username }}"
      key: "{{ item.ssh_key }}"
      state: present
    loop: "{{ users_to_create }}"
    when: item.ssh_key is defined
    tags: [ssh, users]

  - name: "Add '{{ item.username }}' to sudoers"
    lineinfile:
```

```
      path: /etc/sudoers.d/{{ item.username }}
      create: yes
      mode: '0440'
      line: "{{ item.username }} ALL=(ALL) NOPASSWD:ALL"
    when: item.sudo | default(false)
    loop: "{{ users_to_create }}"
    tags: [sudo, users]

- name: "Remove user '{{ item }}'"
    user:
      name: "{{ item }}"
      state: absent
      remove: yes
    loop: "{{ users_to_remove }}"
    tags: [delete, users]
```

✅ What This Playbook Does

	Action
✅	Creates specified users with custom shells
✅	Installs their SSH keys for secure login
✅	Grants sudo access when requested
✅	Removes unwanted users cleanly (with home dirs)

🧠 Why This Task is DevOps-Friendly

- **Idempotent** — rerunning it won't break existing users
- **Modular** — add/remove users via `group_vars/all.yml` only
- **Secure** — handles sudo access through dedicated files
- **Extensible** — easy to integrate into CI/CD pipelines or cloud-init bootstraps

🎯 Real time Task 7 :

Use Ansible to:

- Perform safe and controlled **unattended OS patching**
- **Detect** and **notify** if a reboot is required
- Allow scheduling during **maintenance windows**

✅ Why This Matters in Real-World Environments

- **Security compliance** (e.g., CIS, PCI-DSS, HIPAA)
- Reduce human error during updates
- Avoid unplanned downtime with **controlled reboots**
- Schedule updates in **low-traffic windows**

📋 Task Breakdown

1. Run safe updates (`apt` or `yum`)
2. Notify/report if a reboot is required
3. Optionally reboot (manually or automatically)
4. Ensure idempotency and easy logging

🛠️ Modules Used

- `apt` or `yum` – Perform upgrades
- `shell` or `stat` – Check for reboot-required files
- `reboot` – Handle reboots (optionally)
- `ansible_facts` – Access system info
- `cron` or `at` (optional) – Schedule updates in windows

🧾 Example Playbook (Ubuntu / Debian)

```
---

- name: Patch Management - Safe OS Updates

  hosts: all

  become: yes

  gather_facts: yes


  vars:

    auto_reboot: false

    maintenance_window: "02:00"


  tasks:


    - name: Update package cache

      apt:

        update_cache: yes

        cache_valid_time: 3600
```

```
- name: Upgrade all packages safely
  apt:
    upgrade: safe

- name: Check if reboot is required (Debian/Ubuntu)
  stat:
    path: /var/run/reboot-required
  register: reboot_required

- name: Notify if reboot is needed
  debug:
    msg: "🔁 Reboot is required!"
  when: reboot_required.stat.exists

- name: Reboot the machine (optional)
  reboot:
    msg: "Reboot initiated by Ansible after patching"
    pre_reboot_delay: 30
    reboot_timeout: 600
```

```
      when: reboot_required.stat.exists and auto_reboot


    - name: Log result to local file (optional)
      local_action:
        module: copy
        content: |
          Host: {{ inventory_hostname }}
          Reboot Needed: {{ reboot_required.stat.exists }}
          Updated At: {{ ansible_date_time.date }} {{
ansible_date_time.time }}
        dest: "./patch_logs/{{ inventory_hostname }}.log"
```

🔐 RHEL/CentOS Variant (`yum`)

Replace the `apt` tasks with:

```
- name: Upgrade all packages on RHEL/CentOS
  yum:
    name: "*"
    state: latest
```

And check for reboot:

```
- name: Check if reboot hint exists (RHEL)
  stat:
    path: /var/run/reboot-required
  register: reboot_required
```

Or use:

```
- name: Check if last kernel version is in use
  shell: needs-restarting -r
  register: reboot_hint
  failed_when: false
```

⏱️ Optional: Patch Only in Maintenance Windows

Use `cron` or `at` to delay patch tasks (or run playbook only if `ansible_date_time.hour == "02"`).

📩 Optional: Reporting

Use `mail`, `slack`, or `uri` module to send patch results to your team. Example:

```
- name: Send notification to Slack
  uri:
    url: https://hooks.slack.com/services/XXX
    method: POST
    body: '{"text": "Reboot required on {{ inventory_hostname }}"}'
    headers:
      Content-Type: "application/json"
  when: reboot_required.stat.exists
```

💡 Best Practices

- Tag playbooks with `patch` for easy scheduling:
  `ansible-playbook patch.yml --tags "patch"`
- Use inventory groups like `[prod]`, `[staging]` for phased rollouts
- Store logs centrally (e.g., in S3, log server)

🎯 Goal of This Task

This Ansible task automates the **entire OS patching process** in a clean, controlled, and auditable way. It's especially useful in environments where:

- Systems must stay **secure and up-to-date**
- Reboots need to be **tracked and minimized**
- **Maintenance windows** must be respected
- Patching should happen **without manual SSH login**

🧩 Step-by-Step Explanation

1. Update the Package Cache

```
- name: Update package cache
  apt:
    update_cache: yes
    cache_valid_time: 3600
```

🔍 **Why?**

Before upgrading, we make sure the system knows about the latest package versions. This avoids stale package data, which could cause failed upgrades.

2. Perform a Safe Upgrade

```
- name: Upgrade all packages safely
  apt:
    upgrade: safe
```

🔍 **What is “safe”?**

- Only upgrades **already installed** packages
- Doesn’t remove packages or install new dependencies
- Less risky than `full-upgrade` or `dist-upgrade`

✅ **Real-world Benefit**: This keeps systems secure without unexpected service disruptions.

3. Check If a Reboot Is Required

```
- name: Check if reboot is required (Debian/Ubuntu)
  stat:
    path: /var/run/reboot-required
  register: reboot_required
```

🔍 **Why check this?**
Many Linux systems drop a file here after kernel updates or critical patches — it's a **standard signal** that a reboot is needed for changes to take effect.

4. Notify If Reboot Is Needed

```
- name: Notify if reboot is needed
  debug:
    msg: "🔁 Reboot is required!"
  when: reboot_required.stat.exists
```

🔔 This will log or print a message if a reboot is necessary. In real usage, you might integrate this with Slack, email, or another alerting system.

5. (Optional) Reboot the Server

```
- name: Reboot the machine (optional)
  reboot:
    msg: "Reboot initiated by Ansible after patching"
    pre_reboot_delay: 30
    reboot_timeout: 600
  when: reboot_required.stat.exists and auto_reboot
```

🚨 **Controlled reboot**

- Only happens if a reboot is required **and** you've explicitly allowed it with `auto_reboot: true`
- Waits 30 seconds, gives system time to close services
- Waits up to 600 seconds (10 minutes) for server to come back

✅ Great for **automated but safe** reboot workflows

6. (Optional) Log the Result

```
- name: Log result to local file (optional)
  local_action:
    module: copy
    ...
```

📝 Saves patch and reboot information in a file on the control node. Useful for:

- Audits
- Troubleshooting
- Compliance reporting

⚙️ Optional Enhancements

⏱️ Time-Windowed Patching

This task can easily be adapted to **only run during specific hours** (e.g., 2–3 AM) using:

- `cron` to schedule the playbook

- `when:` conditions based on system time (e.g., `ansible_date_time.hour == '02'`)

📢 Notifications

Add `uri` or `mail` modules to send alerts if:

- A reboot is required
- Patching fails
- Patching completes successfully

✅ Best Practices for Ansible Playbooks & Modules

Ansible is powerful, but with great power comes great responsibility. Following best practices ensures that your automation is:

- **Readable**
- **Reusable**
- **Reliable**
- **Idempotent**
- **Easy to scale & debug**

📘 1. Keep Playbooks Readable and Modular

🔹 **Use roles to organize logic**

Break playbooks into reusable **roles** for tasks like `nginx`, `users`, `mysql`, etc.

```
site.yml
roles/
  ├── nginx/
  ├── users/
```

```
  └── mysql/
```

🔹 **Separate configuration (vars) from logic (tasks)**

Bad:

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

Good:

```
vars:
  nginx_package: nginx

tasks:
  - name: Install nginx
    apt:
      name: "{{ nginx_package }}"
      state: present
      update_cache: yes
```

🧪 2. Write Idempotent Tasks

Ansible is declarative. Every task should be safe to **run multiple times** without causing change.

✅ **Good (idempotent):**

```
- name: Add user
  user:
    name: deploy
    state: present
```

❌ **Bad (not idempotent):**

```
- name: Create user with shell
  shell: useradd deploy
```

📦 3. Use Ansible Modules Over Shell Commands

Always prefer Ansible modules — they're idempotent, easier to read, and error-aware.

✅ **Good:**

```
- name: Install packages
  apt:
```

```
    name: "{{ item }}"
    state: present
  loop:
    - git
    - curl
```

❌ **Bad:**

```
- name: Install packages (bad)
  shell: apt install -y git curl
```

🧩 4. Variable Management

- Use **defaults/main.yml** in roles for sane defaults
- Use **group_vars/host_vars** for targeted overrides
- Avoid hardcoded values
- Use `vars_prompt` for sensitive or interactive input when necessary

🗃️ 5. Inventory & Environment Structure

Maintain clear environment-based inventories:

```
inventories/
  production/
    hosts.yml
    group_vars/
  staging/
    hosts.yml
    group_vars/
```

Use `--inventory` to target environment.

🔁 6. Use Loops and Conditionals Wisely

✅ Example: Looping with `when`

```
- name: Install only if on Debian
  apt:
    name: nginx
    state: present
```

```
  when: ansible_os_family == "Debian"
```

🔒 7. Handle Secrets Securely

Use **Ansible Vault** for passwords, tokens, SSH keys:

```
ansible-vault encrypt secrets.yml
```

```
- name: Use encrypted secret
  debug:
    msg: "My password is {{ vault_db_pass }}"
```

🧪 8. Test Your Playbooks

- Use `--check` for dry runs
- Use `--diff` to show changes
- Use `ansible-lint` to catch bad patterns
- Test playbooks in **Vagrant**, **Docker**, or **EC2 test environments**

📄 9. Use Meaningful Names and Tags

- Give every task a **clear name**
- Use `tags` to run parts of a playbook:

```
- name: Install nginx
  apt:
    name: nginx
    state: present
  tags: nginx
```

Run: `ansible-playbook site.yml --tags nginx`

📑 10. Documentation Inside Your Playbooks

- Use `name:` fields everywhere
- Use comments only when the logic isn't obvious
- Document `README.md` per role

📈 11. Limit Host Impact and Use Serial

- Use `serial` for rolling changes:

```
- hosts: webservers
  serial: 2
```

This prevents downtime across all servers during patching or deployments.

🧠 Bonus Tips:

🔐 1. `become: true` Only Where Needed

✅ **Best Practice:**

Use privilege escalation (`become: true`) **only** on tasks that absolutely require root access.

```
- name: Install NGINX
  become: true
  apt:
    name: nginx
    state: present
```

⚠️ **Why it matters:**

- Improves **security posture**
- Prevents **accidental destructive changes** (especially when variables are misconfigured)
- Aligns with the principle of **least privilege**

🔸 Apply `become: true` at the task level (not globally), unless all tasks in the play require it.

🧪 2. Use `check_mode:` in CI

✅ **Best Practice:**

Run your playbooks in `--check` mode during testing pipelines.

```
ansible-playbook site.yml --check
```

You can also selectively **disable check mode** for specific tasks:

```
- name: Restart service (skip in check mode)
  service:
    name: nginx
    state: restarted
  check_mode: no
```

🎯 **Why it matters:**

- Validates **syntax and structure** without making real changes
- Ensures **idempotency** (Ansible shows what *would* change)

- Great fit for **CI/CD pipelines**, especially when automating infrastructure

🧱 3. Use `block`, `rescue`, `always` for Task Control

This allows **error handling**, much like `try/except/finally` in code.

```
- name: Install with error handling
  block:
    - name: Try installing custom package
      apt:
        deb: /tmp/custom-package.deb
  rescue:
    - name: Fallback to APT install
      apt:
        name: fallback-package
        state: present
  always:
    - name: Clean temp files
      file:
        path: /tmp/custom-package.deb
        state: absent
```

🔍 **Why it matters:**

- **Gracefully handles errors** without breaking the entire playbook
- Ensures **cleanup always runs**
- Allows you to **attempt multiple strategies** (like retrying a failed download)

♻️ 4. Reuse with `include_tasks` and `import_tasks`

Use task inclusion to **break large playbooks into logical units**.

```
# In main.yml
- name: Install web stack
  import_tasks: web.yml

- name: Set up users
  include_tasks: users.yml
```

🔸 **`import_tasks`: Static include (resolved at parse time)**

🔸 **`include_tasks`: Dynamic include (resolved at runtime, can use variables)**

✅ **Why it's powerful:**

- Encourages **DRY (Don't Repeat Yourself)** principles
- Makes **debugging easier** with smaller, modular files
- Promotes **task reusability** across multiple playbooks

> 💡 Use `include_tasks` when task names depend on conditionals or variables.

🌐 5. Store Reusable Roles in Ansible Galaxy or Private Git Repos

- Package reusable logic into **roles**
- Host them in:
    - Ansible Galaxy (`ansible-galaxy publish`)
    - Private Git repos (`requirements.yml`)

```
# requirements.yml
- src: git+https://github.com/my-org/ansible-role-nginx.git
  name: nginx
  version: v1.0.3
```

Install with:

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

🚀 **Why this scales:**

- Makes **roles versioned and maintainable**
- Promotes **collaboration** in DevOps teams
- Easy to **pin, update, or reuse** in other environments/projects

🛠️ Troubleshooting Tips in Ansible

Whether you're working on a single-node test or orchestrating across dozens of production hosts, issues happen. The key is knowing **how to trace, diagnose, and fix** them quickly.

🔍 1. Use `-v`, `-vv`, or `-vvv` for Verbose Output

Ansible gives **minimal output by default**, which is great for clean logs but not for debugging.

- `-v`: shows task-level info (like changes made)
- `-vv`: shows module arguments & connection details
- `-vvv`: shows SSH-level debug, raw output

bash

CopyEdit

```
ansible-playbook site.yml -vvv
```

🔎 Start with `-vv` for most issues.

📂 2. Check the Inventory File

Common issue: hosts not found, or groups incorrectly targeted.

- Ensure your `inventory.ini` or `hosts.yaml` is properly formatted
- Confirm hostnames/IPs are reachable
- Validate group names match the playbook

```
ansible-inventory --list -y
```

✅ Pro Tip: Use `--limit` to narrow scope while testing:

```
ansible-playbook site.yml --limit web1.example.com
```

🧪 3. Use `--check` Mode for Dry Runs

The `--check` flag simulates the playbook without making changes. It helps **validate logic, task flow, and conditional behavior**.

```
ansible-playbook nginx.yml --check
```

Limitations:

- Not all modules support check mode

- Doesn't show file/template content differences

🧼 4. Clean Up Facts & Cache

Issues from outdated facts can confuse playbooks.

🧹 **Clear fact cache:**

```
ansible all -m setup --flush-cache
```

Or disable fact caching in your config or playbook:

```
gather_facts: no
```

> Tip: Only gather facts if you **need them**.

📑 5. Use `ansible-lint` to Catch Common Errors

`ansible-lint` identifies bad practices, typos, or dangerous usage.

```
ansible-lint playbook.yml
```

- Flags deprecated modules
- Warns on shell usage when a module exists

- Ensures consistent syntax and style

🚒 6. Debug Variables and Task Flow

Use the `debug` module:

```
- debug:
    var: some_variable
```

Or print a message:

```
- debug:
    msg: "Deploying to {{ inventory_hostname }}"
```

Use `when:` to narrow down tasks and confirm logic is working.

🧠 7. Trace with Tags and `--start-at-task`

- Use `--tags` to run a subset of tasks
- Use `--start-at-task` to resume long playbooks without re-running everything

```
ansible-playbook site.yml --start-at-task="Install NGINX"
```

🧱 8. Check Permissions & `become` Issues

Common problems:

- Task fails with `Permission denied`
- Missing `become: true` or `become_user`

Example:

```
- name: Restart service
  service:
    name: nginx
    state: restarted
  become: true
```

Also check:

- User in inventory (`ansible_user`)
- SSH key permissions
- `sudo` access for remote user

🔧 9. Common Error Examples & Fixes

Error Message	Likely Cause	Fix
`UNREACHABLE!`	SSH/auth failure	Check IP, user, key, port
`FAILED! => ... no attribute`	Bad variable reference	Check spelling, default/fallback
`File not found`	Relative path problem	Use full path or `{{ role_path }}/files/filename`
`Missing required parameters`	Incomplete module args	Refer to module docs
`Permission denied`	No root access	Use `become: true`, verify sudoers

📚 10. Use `ansible-playbook --syntax-check`

Catches YAML syntax issues or malformed playbooks:

```
ansible-playbook site.yml --syntax-check
```

🧭 11. Inspect Host-Specific Facts

Sometimes a task behaves differently per host. Use the `setup` module to inspect:

```
ansible web1 -m setup | less
```

Search for specific facts:

```
ansible web1 -m setup -a 'filter=ansible_distribution'
```

🧠 12. Check Ansible Configuration

Run this to see **current config, overrides, and paths**:

```
ansible-config dump --only-changed
```

Useful to debug:

- Inventory plugin issues

- Callback plugin behavior
- Fact caching settings
- SSH control settings

🎯 Final Thoughts & What's Next

Congratulations—you've reached the end of 🚀 **Automate Like a Pro: The DevOps Guide to Ansible Playbooks & Modules**!

Over the course of this doc, you've:

- Gained deep understanding of **core Ansible concepts** and **structure**
- Built practical, **real-world automation tasks** that DevOps engineers use daily
- Learned **best practices** to make your playbooks clean, secure, and reusable
- Added **debugging and troubleshooting** techniques that empower you in production

Thank you sincerely for reading this doc—and a massive shout-out to everyone who followed my previous tutorials. Your trust, feedback, and encouragement have inspired each update and pushed this content forward. 🙏

🔭 What to Look Forward To

This guide is only one part of a much larger journey—here's what's coming soon:

- **Advanced Ansible Techniques**: Roles, Collections, Plugins
- **CI/CD Integrations**: GitLab, Jenkins, GitHub Actions
- **Container & Orchestration Workflows**: Docker Compose, Kubernetes
- **Cloud-Native Patterns**: AWS/Azure/GCP provisioning with infrastructure as code
- **Security & Compliance Automation**: Vault, hardening pipelines, audit-ready systems

Stay tuned—there's more powerful, hands-on Ansible wisdom coming your way very soon!

💬 Fenil Gajjar :

> **"Knowledge grows only when shared. Let's walk together, grow together, and lead by example."**