BY FENIL GAJJAR

ANSIBLE DAILY TASKS

ANSIBLE INVENTORY MANAGEMENT

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

CONTACT US :
fenilgajjar.devops@gmail.com
linkedin.com/in/fenl-gajjar
github.com/Fenil-Gajjar

📦 The DevOps Guide to 🚀 Ansible Inventory Unleashed 🧩

👋 Welcome to the Ansible Inventory Management Guide

Hello DevOps enthusiasts!
Welcome to this in-depth, practical guide on **Inventory Management in Ansible** — a fundamental yet often underestimated piece of the automation puzzle.

This document isn't just about theory — it's built for **real-world DevOps engineers**, cloud practitioners, and automation fans like you who want to **understand, implement, and scale** Ansible inventories the right way.

🚀 What to Expect

In the pages ahead, you'll find:

- Easy-to-understand **concepts**
- Hands-on **practical tasks** inspired by real project scenarios
- Clean, YAML-based inventory structures
- Coverage of **static**, **dynamic**, and **mixed-OS** environments
- Industry-backed **best practices** to build robust, scalable automation

Whether you're just getting started or fine-tuning enterprise-grade inventories, this guide will meet you where you are.

🙏 A Note of Thanks

Before we dive in — a heartfelt thank you to **everyone who has supported me on this journey**.

Your encouragement, feedback, and curiosity have inspired me to build and share this resource. I hope it helps you in your journey just as much as you've helped me in mine.

So, let’s dive into the world of Ansible inventory — the place where your infrastructure gets its voice.
📘 *Let’s automate with clarity, confidence, and best practices.*

🧩 What is Inventory in Ansible?

In Ansible, **Inventory** refers to the **list of nodes (hosts/servers)** that Ansible manages and automates. It's the **source of truth** that tells Ansible *which machines to connect to*, *how to connect to them*, and *how to group them for efficient management*.

Think of it as a dynamic phonebook of your infrastructure.

✅ Key Concepts:

- **Inventory** = **Hosts** + **Groups** + **Variables**
- It defines the **target infrastructure**: which hosts, their IPs, SSH ports, usernames, etc.
- It can be written in **INI, YAML**, or dynamically generated via scripts or plugins.

🛠️ Where is Inventory Used?

When running **ad-hoc commands**:

```
ansible all -i inventory.ini -m ping
```

When running **playbooks**:

```
ansible-playbook -i inventory.yml site.yml
```

🔍 **Example (INI format):**

```
[web]

web01 ansible_host=192.168.1.10 ansible_user=ubuntu

web02 ansible_host=192.168.1.11 ansible_user=ubuntu


[db]

db01 ansible_host=192.168.1.20 ansible_user=postgres


[all:vars]

ansible_ssh_private_key_file=~/.ssh/id_rsa
```

- Groups: `[web]`, `[db]`
- Host-level variables: `ansible_host`, `ansible_user`
- Global variables: under `[all:vars]`

🔄 Types of Inventories:

1. **Static Inventory** – Hardcoded host list (INI or YAML format)

2. **Dynamic Inventory** – Generated from cloud APIs (AWS, Azure, GCP, etc.)

📦 Why is it Important?

- Lets you **organize infrastructure by roles**, such as `web`, `db`, `loadbalancer`, etc.

- Enables **environment separation** (dev, staging, prod).

- Inventory design impacts **how playbooks scale and perform**.

🚀 Why Inventory is the Core of Configuration Management in Ansible

Inventory is **the backbone** of how Ansible understands and interacts with your infrastructure. Without inventory, **Ansible has no context** — it doesn't know *what* to configure, *where* to connect, or *how* to apply tasks.

Let's break down why it's so fundamental:

🧠 Defines the Target Infrastructure

Inventory explicitly tells Ansible:

- **Which hosts** are part of the infrastructure
- **How to connect** to each host (IP, SSH user, port, key)
- **What groups** they belong to (e.g., `web`, `db`, `prod`, `staging`)
- Any **host-specific or group-specific variables**

Without this information, your playbooks are just recipes with no ingredients.

🏗️ Enables Scalable Configuration

Inventory allows you to:

- Manage **hundreds or thousands of hosts** with just a few lines
- Apply configurations based on **roles, locations, environments**
- Control **rolling updates**, **conditional logic**, and **host selection**

This scalability is essential in large-scale infrastructure.

🔄 Dynamic Adaptability

With dynamic inventory:

- Your infrastructure can **scale up/down automatically**
- Ansible queries cloud APIs (e.g., AWS EC2, Azure VM) in real time
- You don't need to manually update inventory files

This is critical in cloud-native and containerized environments.

🧩 Drives Variable Management

Inventory isn't just a host list — it's a place to **store and organize variables**, like:

- Hostnames, IPs, connection details
- Application ports, database credentials
- Environment-specific overrides

This tight integration with `group_vars` and `host_vars` makes configuration **clean, reusable, and secure**.

📊 Core for Targeting and Execution

Every Ansible command — whether ad-hoc or via playbook — begins with a question:
 "Which hosts should I run this against?"

Inventory answers that.
 Examples:

```
ansible web -i inventory.ini -m ping

ansible-playbook -i inventory.yml deploy.yml --limit db
```

Without the inventory, these commands are meaningless.

🧭 Types of Inventories in Ansible: Static vs Dynamic

In Ansible, inventories come in two main forms — **Static** and **Dynamic**. Both serve the same purpose (defining the hosts Ansible will manage), but they're suited for different kinds of environments.

Let's explore each type with explanations, examples, and real-world use cases.

🧱 Static Inventory

📌 What is it?

A **manually defined list** of hosts and groups, written in formats like **INI** or **YAML**.

📂 Common file locations:

- `inventory.ini`
- `inventory.yml`
- Passed directly using `-i` flag in commands

🔍 **Example: INI Format**

```
[web]

web01 ansible_host=192.168.1.10

web02 ansible_host=192.168.1.11


[db]

db01 ansible_host=192.168.1.20


[all:vars]

ansible_user=ubuntu

ansible_ssh_private_key_file=~/.ssh/id_rsa
```

🔍 **Example: YAML Format**

```
all:

  children:

    web:

      hosts:

        web01:

          ansible_host: 192.168.1.10
        web02:
          ansible_host: 192.168.1.11
    db:
      hosts:
        db01:
          ansible_host: 192.168.1.20
  vars:
    ansible_user: ubuntu
    ansible_ssh_private_key_file: ~/.ssh/id_rsa
```

✅ **Best for:**

- Small to medium-sized environments
- Static infrastructure (bare metal, small VMs)
- Dev/testing environments

❌ **Limitations:**

- Needs **manual updates** when hosts change

- Doesn’t scale well in **cloud or dynamic environments**

☁️ Dynamic Inventory

📌 What is it?

An **auto-generated list of hosts**, pulled in real-time from **external sources** like cloud providers, CMDBs, or APIs.

Instead of hardcoding host entries, Ansible uses **inventory plugins** or scripts to **query APIs** and get the latest data.

🔧 How it works:

- You define a **plugin config file** (usually YAML)
- Ansible pulls live data from sources like AWS, GCP, Azure, VMware, etc.
- Supports **filters**, **host variables**, **tags**, etc.

🔍 Example: AWS EC2 Dynamic Inventory

```
plugin: aws_ec2

regions:

  - us-east-1

filters:
  tag:Environment: production
keyed_groups:
  - key: tags.Name
hostnames:
  - public-ip-address
```

This fetches all EC2 instances in `us-east-1` with the `Environment=production` tag.

✅ **Best for:**

- **Cloud-native environments**
- **Auto-scaling groups**
- Infrastructure with frequent changes
- Multi-region or multi-cloud architectures

🔗 **Supported sources:**

- AWS (`aws_ec2`)
- Azure (`azure_rm`)

- GCP (`gcp_compute`)
- Kubernetes (`k8s`)
- VMware (`vmware_vm_inventory`)
- Custom scripts (using `script` plugin)

❌ **Limitations:**

- Requires **initial setup** and authentication configs
- May introduce **latency** when fetching large inventories
- Requires Ansible >= 2.8 for full plugin support

🧾 Inventory File Basics in Ansible

(How to write and structure a static inventory file)

In Ansible, the inventory file is the **starting point** of every automation — it tells Ansible *which hosts* to manage and *how to group and configure them*. A static inventory is typically written in **INI** or **YAML** format, with **YAML being the more modern and flexible choice**.

📂 Where Is the Inventory File Located?

You can:

Pass it explicitly using the `-i` option:

```
ansible-playbook -i inventory.yml site.yml
```

Set a **default inventory file** in `ansible.cfg`:

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

Typical file locations:

```
project/
├── inventory/
│   ├── inventory.yml
│   ├── group_vars/
│   └── host_vars/
├── playbooks/
└── ansible.cfg
```

🗂️ INI Format (Legacy, Simple)

A traditional INI inventory looks like this:

```
[web]
web01 ansible_host=192.168.1.10 ansible_user=ubuntu
web02 ansible_host=192.168.1.11 ansible_user=ubuntu

[db]
db01 ansible_host=192.168.1.20 ansible_user=postgres
```

```
[all:vars]

ansible_ssh_private_key_file=~/.ssh/id_rsa
```

✅ **Pros**: Simple, quick to write
❌ **Cons**: Not very expressive or scalable for complex setups

💡 YAML Format (Modern, Recommended)

YAML allows for **structured, nested**, and **more readable** inventory files. It's supported since Ansible 2.4+ and is perfect for advanced use cases.

🔹 **Basic Example:**

```
all:

  hosts:

    web01:

      ansible_host: 192.168.1.10

      ansible_user: ubuntu

    web02:

      ansible_host: 192.168.1.11

      ansible_user: ubuntu

  children:
    db:
      hosts:
        db01:
          ansible_host: 192.168.1.20
          ansible_user: postgres
```

- `all`: root group, required
- `hosts`: individual hosts
- `children`: nested groups like `web`, `db`, etc.

🔹 **With Group Variables:**

```
all:
  children:
    web:
      hosts:
        web01:
        web02:
      vars:
        ansible_user: ubuntu

        app_port: 8080


    db:

      hosts:

        db01:

      vars:

        ansible_user: postgres

        db_port: 5432
```

🔹 **With Host Variables:**

```
all:

  hosts:

    web01:

      ansible_host: 192.168.1.10

      ansible_port: 2222

      ansible_user: deployer
```

📁 4. Directory Structure Best Practices

For scalable environments, organize your inventory like this:

```
inventory/
├── inventory.yml
├── group_vars/
│   ├── web.yml       # Variables for [web] group
│   └── db.yml        # Variables for [db] group
├── host_vars/
│   ├── web01.yml     # Variables specific to web01
│   └── db01.yml
```

This lets you **separate config from structure**, making it easier to manage complex setups.

🧪 Test and Visualize Your Inventory

Use the `ansible-inventory` command:

```
# Outputs the entire inventory in JSON format (expanded view)

ansible-inventory -i inventory.yml --list


# Displays a visual tree-like structure of groups and hosts

ansible-inventory -i inventory.yml --graph
```

🛠️ Task: Define a Basic YAML Inventory for a Master-Worker Infrastructure on AWS

📌 Scenario Overview:

You're managing a small cloud-based Kubernetes-like infrastructure on **AWS**, consisting of:

- **1 Master Node** – responsible for orchestration and control
- **3 Worker Nodes** – responsible for running workloads (containers, apps, etc.)

This setup needs to be clearly defined in your **Ansible inventory file**, with proper grouping and variables to support automated provisioning or configuration.

🎯 Objective:

Create a clean, readable, and well-structured **YAML-based static inventory** for this infrastructure. The file should:

- Group all the worker nodes under `workers`
- Include the `master` in its own group

- Provide relevant host-specific details (like `ansible_host`, `ansible_user`)

- Define common variables at the group level

📁 **Directory Assumption:**

We'll assume this inventory is in the following path:

```
inventory/

└── inventory.yml
```

📄 **inventory.yml (YAML Static Inventory File)**

```
all:

  children:


    master:

      hosts:

        master-node:

          ansible_host: 3.91.102.5

          ansible_user: ec2-user
          ansible_ssh_private_key_file: ~/.ssh/aws_key.pem


    workers:

      hosts:

        worker-node-1:

          ansible_host: 3.88.210.23

          ansible_user: ec2-user

          ansible_ssh_private_key_file: ~/.ssh/aws_key.pem


        worker-node-2:

          ansible_host: 3.92.17.44

          ansible_user: ec2-user

          ansible_ssh_private_key_file: ~/.ssh/aws_key.pem


        worker-node-3:

          ansible_host: 3.93.84.112

          ansible_user: ec2-user

          ansible_ssh_private_key_file: ~/.ssh/aws_key.pem
```

```
vars:
  ansible_connection: ssh
  ansible_port: 22
```

🔍 **Explanation of Key Sections**

- **Groups (`master`, `workers`)**: Organize hosts by their function for targeted automation.
- **`ansible_host`**: Public IP of each AWS EC2 instance.
- **`ansible_user`**: Default EC2 user (e.g., `ec2-user` for Amazon Linux).
- **`ansible_ssh_private_key_file`**: Path to your SSH key to connect.
- **Global variables (`all.vars`)**: Apply to every host — SSH connection, default port.

📌 **Bonus Tips:**

You can now run a ping test on just worker nodes:

```
ansible workers -i inventory/inventory.yml -m ping
```

- Or run a full cluster health check using a playbook:

  ```
  ansible-playbook -i inventory/inventory.yml
  playbooks/health-check.yml
  ```

🧩 What Are Inventory Groups and Host Variables?

In Ansible:

- **Groups** are *collections of hosts* with similar roles (e.g. `web`, `db`, `prod`, `workers`).
- **Host variables** are settings that apply *to a specific host*.
- **Group variables** are settings that apply *to every host in that group*.
- You can even **nest groups** using `children`.

Inventory is more than just a list — it's a **blueprint of your infrastructure**, where grouping and variables help you manage with precision and elegance.

🔧 Why Do They Matter?

Using groups and variables:

- Makes **playbooks modular** and **reusable**
- Avoids **repetition** of SSH settings, credentials, or role-based parameters
- Supports **role-specific logic**, **host-level overrides**, and **environmental separation**

✅ Anatomy of a Well-Structured Inventory (YAML Style)

Here's a rich, real-world example of a YAML inventory that demonstrates:

- Group-level variables
- Host-level variables
- Group nesting (via `children`)
- Multiple environments
- Clean separation of logic

📄 Example: Full YAML Inventory Structure

```
all:
  children:


    production:
      children:
        web:
          hosts:
            web1.prod.example.com:
```

```yaml
        ansible_host: 34.201.100.10

        ansible_user: ubuntu

      web2.prod.example.com:

        ansible_host: 34.201.100.11

        ansible_user: ubuntu

    vars:

      app_env: production

      nginx_port: 80


  db:

    hosts:

      db1.prod.example.com:

        ansible_host: 10.0.1.10

        ansible_user: postgres

        db_engine: postgresql

    vars:

      db_port: 5432

      db_user: admin
  staging:
    children:
      web:
        hosts:
          web1.staging.example.com:
            ansible_host: 34.203.100.20
            ansible_user: ubuntu
        vars:
          app_env: staging
          nginx_port: 8080

vars:
  ansible_ssh_private_key_file: ~/.ssh/aws-key.pem
  ansible_connection: ssh
  ansible_port: 22
```

🔍 Let's Break It Down:

🧱 1. `all` Group

This is the **top-most group**, *automatically defined* by Ansible. It includes everything.

yaml

CopyEdit

```yaml
all:
```

🧪 2. Nested Environments: `production` and `staging`

These are **environmental groupings**, commonly used in cloud setups:

```yaml
production:
  children:
    web:
    db:
```

- `production` includes `web` and `db`
- Same structure for `staging`

This lets you run playbooks like:

```
ansible-playbook -i inventory.yml site.yml --limit production
```

💻 3. Host-Specific Variables

These override group or global settings and are used for per-host tweaks:

```
db1.prod.example.com:
  ansible_host: 10.0.1.10
  db_engine: postgresql
```

🧩 4. Group Variables (`vars`)

Define shared settings for a group (no need to repeat on each host!):

```
vars:
  app_env: production
  nginx_port: 80
```

All hosts in the `web` group will inherit this.

🌍 5. Global Variables

Defined under `all.vars`, this applies to **every host** in the inventory:

```
all:
  vars:
    ansible_connection: ssh
    ansible_ssh_private_key_file: ~/.ssh/aws-key.pem
```

Great for:

- SSH key config
- Port setup
- Connection settings

📁 BONUS: External Variables (Cleaner Design)

When your inventory grows, you can move variables out of the inventory file for clarity:

```
inventory/
├── inventory.yml
├── group_vars/
```

```
│   ├── web.yml         # Group-level vars for [web]
│   └── db.yml
├── host_vars/
│   ├── web1.prod.example.com.yml
│   └── db1.prod.example.com.yml
```

Now you can clean up your `inventory.yml` and let Ansible auto-load these variable files.

🎯 Tips for Best Practices

Tip	Why it matters
🔁 Use groups to reduce repetition	One setting for many hosts
⚙️ Use `group_vars/` and `host_vars/` folders	Cleaner, scalable, modular
🔒 Never hardcode secrets in inventory	Use Ansible Vault instead
🔄 Override wisely	Host-level vars override group-level vars
📚 Document groups clearly	Helps collaboration in team environments

---

🧾 Why Use Host Ranges?

Imagine you have 10 worker nodes named like this:

```
worker01, worker02, ..., worker10
```

Instead of writing each one manually, Ansible lets you define them as a **range**, saving time and keeping your inventory DRY (Don't Repeat Yourself).

📌 INI Format (Legacy, but simple)

```
[workers]

worker[01:10].example.com ansible_user=ubuntu
```

`worker[01:10].example.com` will expand to:

```
worker01.example.com

worker02.example.com

...

worker10.example.com
```

✅ Tip: Leading zero (`01`) ensures numbers are zero-padded.

🧾 YAML Format (Recommended)

In YAML, you **can't** directly use the `worker[01:10]` syntax — but you can still keep it readable using a Jinja loop with an **external script or dynamic inventory**, or just define it clearly with list-style formatting.

But for static inventories in YAML, define explicitly like this:

```
all:
  children:
    workers:
      hosts:
        worker01.example.com:
          ansible_user: ubuntu
        worker02.example.com:
          ansible_user: ubuntu
        worker03.example.com:
          ansible_user: ubuntu
        worker04.example.com:
          ansible_user: ubuntu
        worker05.example.com:
          ansible_user: ubuntu
```

🧠 **Pro Tip:** You can generate the YAML snippet programmatically using Python or Bash for dozens/hundreds of hosts.

🛠️ **Automation Trick: Generate with Python**

```
for i in range(1, 11):
    print(f"        worker{str(i).zfill(2)}.example.com:")
    print("          ansible_user: ubuntu")
```

Paste the result directly into your `inventory.yml`.

🧾 Assigning Variables to Hosts and Groups in Ansible

When you're managing infrastructure at scale, you often need to assign variables like:

- SSH user
- Port numbers
- Environment names
- Custom app config
- Credentials (preferably encrypted with Vault)

In Ansible, you assign variables at two key levels:

🎯 1. Group Variables → Apply to Many Machines

Use **group variables** when you want **all hosts in a group** (e.g. all `web` servers or all `db` nodes) to share the same config.

✅ **When to use:** All web servers use `nginx`, or all staging nodes share the same environment variable.

📄 **Example: Assigning Group Variables (Inside Inventory YAML)**

```
all:
  children:
    web:
      hosts:
        web1.example.com:
        web2.example.com:
      vars:
        ansible_user: ubuntu
        app_env: production
        nginx_port: 80
```

All hosts in the `web` group:

- Use the `ubuntu` user
- Serve the production environment
- Run Nginx on port 80

📁 Preferred (Clean) Method: Use `group_vars/`

```
inventory/
├── inventory.yml
└── group_vars/
    └── web.yml
```

group_vars/web.yml:

```
ansible_user: ubuntu
app_env: production
nginx_port: 80
```

🧠 *Ansible automatically loads this when the `web` group is targeted.*

🎯 2. Host Variables → Apply to One Machine

Use **host variables** when a host has **custom settings** that shouldn't affect others.

✅ **When to use:** One server has a different IP, custom SSH port, or database credentials.

📄 Example: Assigning Host Variables

```
all:
  hosts:
    db1.example.com:
      ansible_host: 192.168.10.10
      ansible_user: postgres
      db_engine: postgresql
      db_port: 5432
```

Only `db1.example.com` will receive these values.

📁 Preferred Method: Use `host_vars/`

```
inventory/
├── inventory.yml
└── host_vars/
    └── db1.example.com.yml
```

host_vars/db1.example.com.yml:

```
ansible_user: postgres
db_engine: postgresql
db_port: 5432
```

⚙️ Variable Precedence

Level	Priority
Host variables	🔺 Highest
Group variables	🔽 Lower
`all` group vars	🔽 Even lower
Defaults in role	🔽 Lowest

🔐 Connecting to Hosts: Behavioral Inventory Parameters in Ansible

🎯 What Are Behavioral Inventory Parameters?

Behavioral parameters are **Ansible-specific variables** that control *how* it connects to your hosts — over SSH, WinRM, custom ports, different usernames, private keys, etc.

You assign these variables either:

- In the inventory file
- In `host_vars/` or `group_vars/`
- Or pass them via CLI or playbooks

🧾 Commonly Used Behavioral Parameters

Here's a **comprehensive cheat sheet** of useful behavioral parameters — all with real-world examples.

✅ `ansible_host`

🔹 **Defines the real IP or DNS of the host (if hostname is just a label)**

```
web1:
  ansible_host: 192.168.1.10
```

> Useful when your internal DNS name differs from the SSH target.

✅ `ansible_user`

🔹 **Username to connect with (e.g., `ubuntu`, `ec2-user`, `root`)**

```
ansible_user: ubuntu
```

> Overrides your default local user. Needed for cloud machines.

✅ `ansible_port`

🔹 **SSH Port (default is 22)**

```
ansible_port: 2222
```

> If your SSH service runs on a custom port.

✅ `ansible_ssh_private_key_file`

🔹 **Path to private SSH key**

```
ansible_ssh_private_key_file: ~/.ssh/aws-key.pem
```

> Crucial for connecting to cloud hosts like AWS EC2.

✅ `ansible_connection`

🔹 **Connection type (usually `ssh`, but also `local`, `docker`, `winrm`, etc.)**

```
ansible_connection: ssh
```

> Use `local` for localhost or `docker` when managing containers.

✅ `ansible_become`, `ansible_become_user`

🔹 **Enable privilege escalation (like `sudo`)**

```
ansible_become: true

ansible_become_user: root
```

Needed when you connect as a non-root user but need root privileges.

✅ `ansible_python_interpreter`

🔹 **Set specific Python path (important for non-default distros or environments)**

```
ansible_python_interpreter: /usr/bin/python3
```

Fixes issues when `/usr/bin/python` isn't available or isn't Python 3.

📁 Example YAML Inventory with Behavioral Parameters

```
all:

  children:

    web:

      hosts:
```

```
    web1.example.com:
      ansible_host: 10.0.0.11
      ansible_user: ubuntu
      ansible_port: 22
      ansible_ssh_private_key_file: ~/.ssh/aws-key.pem
      ansible_connection: ssh
      ansible_become: true
      ansible_become_user: root
      ansible_python_interpreter: /usr/bin/python3
```

This setup ensures Ansible connects, escalates privileges, and uses the correct Python on `web1`.

🔍 Testing Host Connectivity

Use the `ping` module to validate everything's working:

```
ansible all -i inventory.yml -m ping
```

Or test a single host:

```
ansible web1.example.com -i inventory.yml -m ping -u ubuntu
```

✅ Best Practices for Inventory Management in Ansible

These best practices apply whether you're managing 10 nodes or 10,000 — across on-prem, cloud, hybrid, or containerized environments.

1️⃣Use YAML (INI is legacy)

Why: YAML is structured, easier to read, and supports nested groups and variables cleanly.

✅ Recommended:

```
all:
  children:
    web:
      hosts:
        web1.example.com:
        web2.example.com:
```

❌ Avoid:

```
[web]
web1.example.com
```

```
web2.example.com
```

YAML promotes consistency and integrates better with modern tools.

2️⃣ Group by Function, Role, or Environment

Use **logical groupings** like:

- `web`, `db`, `cache`
- `dev`, `staging`, `prod`
- `frontend`, `backend`

✅ Example:

```
all:
  children:
    prod:
      children:
        web:
        db:
```

Makes targeting playbooks easy and readable: ***`ansible-playbook -l web site.yml`***

3️⃣Use `group_vars/` and `host_vars/` Instead of Inline Variables

Keep your inventory clean and separate config details.

✅ Recommended:

```
inventory/
├── inventory.yml
├── group_vars/
│   ├── web.yml
│   └── db.yml
├── host_vars/
│   └── db1.example.com.yml
```

Improves clarity, reuse, and security (with Vault).

4️⃣ Use Descriptive Hostnames (Not IPs Directly)

Use DNS names or aliases in inventory — assign IPs via `ansible_host`.

✅ Better:

```
web1:
  ansible_host: 192.168.1.10
```

❌ Avoid:

```
192.168.1.10
```

You gain flexibility if the IP changes.

5️⃣ Use Dynamic Inventory for Cloud Infrastructure

Static inventories are not scalable in cloud-native environments.

✅ Use:

- AWS: `aws_ec2` plugin
- GCP: `gcp_compute`

- Azure: `azure_rm`

```
ansible-inventory -i aws_ec2.yml --list
```

Dynamic inventories sync automatically with your live infra.

6️⃣ Avoid Duplicating Hosts in Multiple Groups

Avoid placing the same host in multiple unrelated groups — it causes variable conflicts and logic bugs.

✅ Better:

- Use nested groups if needed
- Or clearly isolate roles

7️⃣ Always Set the Python Interpreter (Especially for Minimal OS)

Set:

```
ansible_python_interpreter: /usr/bin/python3
```

Prevents `python not found` errors on distros like Ubuntu Minimal, Amazon Linux 2, etc.

8️⃣ Use Vault for Sensitive Variables

Never store plain passwords, keys, or secrets in the inventory.

✅ Use:

```
ansible-vault encrypt_string --name 'db_password'
'SuperSecret123'
```

Or encrypt entire `group_vars/prod.yml`.

9️⃣ Use Behavioral Variables Only Where Needed

Avoid cluttering inventory with repeated values like `ansible_user`, `ansible_port`, etc., if the defaults work.

✅ Use at group level (e.g. for all `web` servers) instead of per-host.

🔟 Document Inventory Structure Clearly

Maintain a README inside your inventory directory explaining:

- Group naming conventions
- Host naming logic

- Inventory split strategy (per env, team, app)

Makes collaboration and onboarding smoother.

🧩 Bonus Tips:

Practice	Why It Matters
Keep inventory in Git	Version control & rollback
Tag servers consistently	Useful with dynamic inventory plugins
Use inventory plugins over scripts	Native, safer, and better maintained
Test inventory syntax	Use `ansible-inventory --graph` or `--list`

🛠️ Task: Configure Host-Specific SSH Ports in Ansible Inventory

🧩 Scenario

You're managing a fleet of Linux servers, and for security reasons, each host uses a **custom SSH port** instead of the default `22`.

Here's the infrastructure:

Hostname	IP Address	SSH Port
`web1.example.com`	`192.168.10.10`	`2222`
`db1.example.com`	`192.168.10.20`	`2200`
`cache1.example.com`	`192.168.10.30`	`2022`

You want Ansible to connect to each host using its correct SSH port.

📁 Step 1: Create YAML Inventory File

```
all:

  children:

    web:

      hosts:

        web1.example.com:

          ansible_host: 192.168.10.10

          ansible_port: 2222

          ansible_user: ubuntu

          ansible_ssh_private_key_file: ~/.ssh/id_rsa_web

    db:

      hosts:

        db1.example.com:

          ansible_host: 192.168.10.20

          ansible_port: 2200

          ansible_user: ubuntu

          ansible_ssh_private_key_file: ~/.ssh/id_rsa_db

    cache:

      hosts:
        cache1.example.com:
          ansible_host: 192.168.10.30
          ansible_port: 2022
          ansible_user: ubuntu
          ansible_ssh_private_key_file: ~/.ssh/id_rsa_cache
```

📋 **Explanation**

- `ansible_host`: The actual IP of the machine
- `ansible_port`: The **custom SSH port** Ansible should use to connect
- `ansible_user`: Login user on the remote machine
- `ansible_ssh_private_key_file`: Key specific to that host

> ⚠️ If you're using the same SSH key for all hosts, you can move `ansible_ssh_private_key_file` to `group_vars/all.yml` instead.

🧪 Step 2: Test Connections

Ping all hosts to ensure Ansible can reach them via the correct port:

```
ansible all -i inventory.yml -m ping
```

Or test a specific group:

```
ansible web -i inventory.yml -m ping
```

🎯 What This Teaches

- How to use host-specific SSH ports with `ansible_port`
- Clean separation of infrastructure by function (web, db, cache)
- How to structure an inventory with real host metadata

🛠️ Task: Define Multiple Environments (dev, staging, prod) in a Single Inventory

📘 Scenario

Your team manages three environments:

- `dev`: For internal development
- `staging`: For QA testing
- `prod`: For live users

Each environment has its own set of app servers and DBs, but **uses the same playbook**. You want to:

- Structure your inventory cleanly by environment
- Make it easy to target specific environments
- Assign shared variables to each environment (like region, user, interpreter)

📁 Step 1: Inventory Layout (`inventory.yml`)

```yaml
all:

  children:

    dev:

      children:

        dev_app:

          hosts:

            dev-app1.example.com:

              ansible_host: 10.0.1.10

            dev-app2.example.com:

              ansible_host: 10.0.1.11

        dev_db:

          hosts:

            dev-db1.example.com:

              ansible_host: 10.0.1.20


    staging:

      children:
        staging_app:
          hosts:
            staging-app1.example.com:
              ansible_host: 10.0.2.10
        staging_db:
          hosts:
            staging-db1.example.com:
              ansible_host: 10.0.2.20

    prod:
      children:
        prod_app:
          hosts:
            prod-app1.example.com:
              ansible_host: 10.0.3.10
            prod-app2.example.com:
              ansible_host: 10.0.3.11
        prod_db:
          hosts:
            prod-db1.example.com:
              ansible_host: 10.0.3.20
```

📁 Step 2: Add `group_vars/` for Each Environment

Create files like:

`group_vars/dev.yml`

```
ansible_user: ubuntu
ansible_python_interpreter: /usr/bin/python3
region: us-west-1
env: dev
```

`group_vars/prod.yml`

```
ansible_user: ansibleadmin
ansible_python_interpreter: /usr/bin/python3
region: us-east-1
env: prod
```

This way, each environment gets its own SSH user, region config, etc., without cluttering the main inventory.

🧪 Step 3: Target Environments in Playbooks

To run your playbook only on `dev`:

```
ansible-playbook -i inventory.yml site.yml -l dev
```

To run only on production DBs:

```
ansible-playbook -i inventory.yml site.yml -l prod_db
```

🎯 What You Learn

- Structuring inventory around environments
- Using nested groups for apps vs databases
- Keeping environment-specific config in `group_vars/`
- Dynamically targeting any layer: full env, app layer, or DB layer

🛠️ Task: Manage a Mixed Environment of Linux and Windows Hosts in a Single Inventory

📘 Scenario

You're managing an environment that includes:

- **Linux servers** for application hosting (Ubuntu or CentOS)
- **Windows servers** for Active Directory and file sharing

You want to:

- Manage both OS types with the **same Ansible inventory**
- Use the right connection methods (`ssh` vs `winrm`)
- Assign correct interpreters, users, ports, and privilege settings

📁 Step 1: Define Inventory (`inventory.yml`)

```yaml
all:
  children:
    linux_servers:
      hosts:
        app01.linux.local:
          ansible_host: 192.168.10.10
        db01.linux.local:
          ansible_host: 192.168.10.11

    windows_servers:
      hosts:
        win01.windows.local:
          ansible_host: 192.168.10.20
        ad01.windows.local:
          ansible_host: 192.168.10.21
```

📁 Step 2: Set Group Variables

`group_vars/linux_servers.yml`

```yaml
ansible_connection: ssh

ansible_user: ubuntu

ansible_become: true

ansible_become_method: sudo

ansible_python_interpreter: /usr/bin/python3
```

`group_vars/windows_servers.yml`

```yaml
ansible_connection: winrm

ansible_user: Administrator

ansible_password: "{{ vault_windows_password }}"

ansible_port: 5986

ansible_winrm_transport: basic

ansible_winrm_server_cert_validation: ignore
```

> 🛡️ **Tip**: Use Ansible Vault for passwords like `vault_windows_password`.

🔐 Step 3: Set Up WinRM on Windows (One-Time)

Use PowerShell on the Windows host to enable WinRM (simplified):

```
winrm quickconfig

winrm set winrm/config/service/Auth @{Basic="true"}

winrm set winrm/config/service @{AllowUnencrypted="true"}

Enable-PSRemoting -Force
```

Or better, use an Ansible role like `ansible-windows-winrm` if bootstrapping remotely.

🧪 Step 4: Test Connectivity

```
ansible all -i inventory.yml -m ping
```

- You should see `pong` from both Linux and Windows hosts
- If any fail, try `-vvv` for verbose output

🧬 Bonus: Mixed OS Playbook Targeting

You can now write OS-specific tasks and use `when` clauses:

```
- name: Mixed OS playbook
  hosts: all
  tasks:
    - name: Linux-specific task
      shell: uname -a
      when: ansible_connection == 'ssh'

    - name: Windows-specific task
      win_command: hostname
      when: ansible_connection == 'winrm'
```

🎯 What You Learn

- How to manage mixed environments cleanly
- Differentiate connection types (`ssh`, `winrm`)
- Set appropriate interpreters, ports, users
- Structure inventory for OS-specific logic
- Use Ansible Vault to protect sensitive data

🧾 Wrap-Up: From Inventory Chaos to Inventory Clarity

And that's a wrap! 🎉
You've just completed a deep, real-world journey through one of the most crucial pillars of Ansible — **Inventory Management**.

In this doc, we explored how to:

- Understand what an inventory is — and why it's the **heart of Ansible**
- Structure clean, scalable inventories using **YAML**
- Use **static and dynamic inventories** effectively
- Apply real-time scenarios like **multi-environment setups** and **mixed OS infrastructure**
- Implement behavioral parameters, best practices, and secure connections
- Bring clarity, order, and professionalism to your infrastructure automation

This wasn't just a theory dump — it was crafted with real project scenarios to help **you implement, not just learn**.

🙏 Thanks for Being Part of This

I truly appreciate your support and time in reading through this guide.
Your encouragement and interest keep this journey going — and I hope this document added real value to your DevOps toolbox.

If this helped you or sparked any new ideas, feel free to share it, drop feedback, or connect further.

🔜 What's Next?

Stay tuned!
The next doc in this series will dive into:

👉 Group & Host Variables in Ansible

- How to assign variables cleanly and flexibly
- The difference between group vars, host vars, and defaults
- Real use-cases and directory structure tips

So if you enjoyed this guide — the next one is going to level you up even more. 🚀

~ ‘ Fenil Gajjar ‘ :

“ Thanks again for walking this path with me.

Let’s keep building clean, efficient, and intelligent automation — one YAML file at a time. 🙌 “