**Ansible Galaxy**

Ansible Galaxy is a community-powered marketplace for Ansible content.

You can download, use, share, or publish roles and collections.

Think of it as the DockerHub for Ansible roles.

URL: <https://galaxy.ansible.com>

**Uses of Roles from Ansible Galaxy**

Roles help you avoid reinventing the wheel.

Example: Installing Docker manually in Ansible is complex due to OS differences (Debian, Ubuntu, RHEL, Alpine, etc.).

Roles in Galaxy are often well-tested across multiple platforms and include necessary conditionals.

Benefit: Saves time, reduces complexity, and promotes best practices.

You can manage roles using the ansible-galaxy CLI :

ansible-galaxy role -h

init: Create a skeleton for a new role.

install: Install a role from Galaxy.

remove: Uninstall a role.

list: List installed roles.

search: Search for roles.

import: Import roles to Galaxy (for publishing).

delete: Delete a role (from local system).

**Publishing Your Own Role to Ansible Galaxy**

**1. Prepare the Role Locally**

Ensure your role has:

* Proper directory structure (tasks, handlers, defaults, meta, etc.)
* A well-written meta/main.yml:

**2. Push Role to GitHub**

cd your\_role\_folder

git init

git remote add origin <your\_repo\_url>

git add .

git commit -m "Initial commit"

git push origin main

**3. Link GitHub to Ansible Galaxy**

* Log in to Ansible Galaxy using your GitHub account.
* Navigate to your namespace (e.g., https://galaxy.ansible.com/your\_namespace).
* Click "My Content" → "Add Content" → Import Role.
* Provide GitHub repo details and import.

**4. Use API Token for CLI Publishing**

Generate an API token from Galaxy under My Profile > API Token

Publish role using CLI:

ansible-galaxy role import <GitHub\_username> <repo\_name> --api-key <your\_token>

Once imported, your role will be visible and usable by others.

**ANSIBLE ROLES**

Imagine writing a single YAML playbook file with 2000+ lines. It becomes:

* Difficult to read
* Harder to maintain
* Practically impossible to reuse

So instead of putting all variables, tasks, handlers, and files in one place, we split them up into structured folders, making the code cleaner, reusable, and more scalable.

**Definition**: A role is just a structured way to organize Ansible content.

If you can write a playbook, you already know 90% of what you need to use roles.

Roles split playbook content into folders like:

|  |  |
| --- | --- |
| tasks/ | Main logic of your automation. All your playbook tasks go here. |
| vars/ | Variables used by the role, higher precedence than defaults. |
| defaults/ | Default variable values (lowest precedence). |
| handlers/ | Special tasks triggered on change (e.g., restart Apache). |
| meta/ | Metadata like author, dependencies, and supported platforms. |
| files/ | Static files to copy to target systems (e.g., HTML, config files). |
| templates/ | Jinja2 templates for dynamic configuration files. |

**Use roles when:**

Your playbook exceeds 100+ lines

You are building complex systems, such as:

Kubernetes (control plane, worker nodes)

Database clusters (Oracle, MySQL, PostgreSQL)

Load balancers, firewalls, etc.

You want to reuse roles across teams/projects

You plan to share roles via Ansible Galaxy or GitHub

ansible-galaxy role init <role-name>

**Ansible Deep-Dive: EC2 Provisioning, Ansible Vault, and Variable Precedence**

**Provisioning EC2 Instances with Ansible & Ansible Vault**

**EC2 Instance Creation Using Ansible**

* Used ansible-playbook to provision an EC2 instance.
* The Playbook interacts with AWS using the boto3 Python library and Ansible AWS collection.
* Required AWS credentials (access key and secret key).

**Ansible Vault (Secret Management)**

* Why Vault?: To secure sensitive data such as AWS credentials.
* The credentials were encrypted using Ansible Vault.
* A .vault\_pass file was created containing the vault password.
* To decrypt and use the secrets:

ansible-playbook ec2\_create.yaml --vault-password-file vault.pass

**Troubleshooting Common Errors**

* Syntax issues are common—check line and column numbers.
* Error due to undefined variables (e.g., EC2 access key).
* Debugging steps: read error trace, fix YAML formatting, ensure variables are passed correctly.

**Result Verification**

* Confirmed EC2 instance creation from the AWS Console.
* The instance was visible in the US-East-1 region.
* Confirmed success by checking instance status and type.

**Ansible Variables and Precedence**

**Why Use Variables in Ansible?**

* Avoid Hardcoding: Makes Playbooks reusable and flexible.
* Instead of hardcoding t2.micro, use a variable like {{ instance\_type }}.

**Where to Define Variables in Ansible?**

Ansible supports 22+ locations for defining variables. You don’t need to memorize them all—just know the most common and important ones:

|  |  |  |
| --- | --- | --- |
| level |  |  |
| 1.(lowest) | defaults/main.yml | Default values defined inside roles. Used when no override is provided. Ideal for DevOps engineers creating Playbooks. |
| 2. | vars/main.yml | Overrides defaults. Used by individual teams or subprojects to change role behavior. |
| 3. | Group Variables (group\_vars/) | Apply variables to groups of hosts (e.g., app, db). Useful for environment-specific or tier-specific configs. |
| 4. | Host Variables (host\_vars/) | Apply variables to individual hosts. |
| 5. | Playbook-level Vars | Inline vars: within a Play. Limited to that Play. |
| 6.(highest) | Extra Vars (--extra-vars or -e) | Supplied via command-line during Playbook execution. Always takes top priority. |

**Best Practices**

* Use defaults/ for base configurations (DevOps engineers).
* Use vars/ for team-specific overrides.
* Use group\_vars/ or host\_vars/ for environment-specific settings.
* Use --extra-vars only for testing or quick overrides (not for long-term config).
* Avoid hardcoding values directly in the Playbook.
* Use Jinja2 templating ({{ variable\_name }}) to reference variables.