Tips and tricks

These tips and tricks have helped us optimize our Ansible usage, and we offer them here as suggestions. We hope they will help you organize content, write playbooks, maintain inventory, and execute Ansible. Ultimately, though, you should use Ansible in the way that makes most sense for your organization and your goals.

General tips

These concepts apply to all Ansible activities and artifacts.

Keep it simple

Whenever you can, do things simply. Use advanced features only when necessary, and select the feature that best matches your use case. For example, you will probably not need `vars`, `vars_files`, `vars_prompt` and `--extra-vars` all at once, while also using an external inventory file. If something feels complicated, it probably is. Take the time to look for a simpler solution.

Use version control

Keep your playbooks, roles, inventory, and variables files in git or another version control system and make commits to the repository when you make changes. Version control gives you an audit trail describing when and why you changed the rules that automate your infrastructure.

Customize the CLI output

You can change the output from Ansible CLI commands using :ref:`callback_plugins`.

> **System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_best_practices.rst`, line 31); *backlink***
>
> Unknown interpreted text role "ref".

Playbook tips

These tips help make playbooks and roles easier to read, maintain, and debug.

Use whitespace

Generous use of whitespace, for example, a blank line before each block or task, makes a playbook easy to scan.

Always name tasks

Task names are optional, but extremely useful. In its output, Ansible shows you the name of each task it runs. Choose names that describe what each task does and why.

Always mention the state

For many modules, the 'state' parameter is optional. Different modules have different default settings for 'state', and some modules support several 'state' settings. Explicitly setting 'state=present' or 'state=absent' makes playbooks and roles clearer.

Use comments

Even with task names and explicit state, sometimes a part of a playbook or role (or inventory/variable file) needs more explanation. Adding a comment (any line starting with '#') helps others (and possibly yourself in future) understand what a play or task (or variable setting) does, how it does it, and why.

Inventory tips

These tips help keep your inventory well organized.

Use dynamic inventory with clouds

With cloud providers and other systems that maintain canonical lists of your infrastructure, use :ref:`dynamic inventory <intro_dynamic_inventory>` to retrieve those lists instead of manually updating static inventory files. With cloud resources, you can use tags to differentiate production and staging environments.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_best_practices.rst`, line 67); *backlink*
>
> Unknown interpreted text role "ref".

Group inventory by function

A system can be in multiple groups. See :ref:`intro_inventory` and :ref:`intro_patterns`. If you create groups named for the function of the nodes in the group, for example *webservers* or *dbservers*, your playbooks can target machines based on function. You can assign function-specific variables using the group variable system, and design Ansible roles to handle function-specific use cases. See :ref:`playbooks_reuse_roles`.

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_best_practices.rst`, line 72); *backlink*
>
> Unknown interpreted text role "ref".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_best_practices.rst`, line 72); *backlink*
>
> Unknown interpreted text role "ref".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_best_practices.rst`, line 72); *backlink*
>
> Unknown interpreted text role "ref".

Separate production and staging inventory

You can keep your production environment separate from development, test, and staging environments by using separate inventory files or directories for each environment. This way you pick with -i what you are targeting. Keeping all your environments in one file can lead to surprises! For example, all vault passwords used in an inventory need to be available when using that inventory. If an inventory contains both production and development environments, developers using that inventory would be able to access production secrets.

Keep vaulted variables safely visible

You should encrypt sensitive or secret variables with Ansible Vault. However, encrypting the variable names as well as the variable values makes it hard to find the source of the values. To circumvent this, you can encrypt the variables individually using `ansible-vault encrypt_string`, or add the following layer of indirection to keep the names of your variables accessible (by `grep`, for example) without exposing any secrets:

1. Create a `group_vars/` subdirectory named after the group.
2. Inside this subdirectory, create two files named `vars` and `vault`.

3. In the `vars` file, define all of the variables needed, including any sensitive ones.
4. Copy all of the sensitive variables over to the `vault` file and prefix these variables with `vault_`.
5. Adjust the variables in the `vars` file to point to the matching `vault_` variables using jinja2 syntax: `db_password: {{ vault_db_password }}`.
6. Encrypt the `vault` file to protect its contents.
7. Use the variable name from the `vars` file in your playbooks.

When running a playbook, Ansible finds the variables in the unencrypted file, which pulls the sensitive variable values from the encrypted file. There is no limit to the number of variable and vault files or their names.

Note that using this strategy in your inventory still requires *all vault passwords to be available* (for example for `ansible-playbook` or AWX/Ansible Tower) when run with that inventory.

Execution tricks

These tips apply to using Ansible, rather than to Ansible artifacts.

Try it in staging first

Testing changes in a staging environment before rolling them out in production is always a great idea. Your environments need not be the same size and you can use group variables to control the differences between those environments.

Update in batches

Use the 'serial' keyword to control how many machines you update at once in the batch. See :ref:`playbooks_delegation`.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_best_practices.rst, line 111); *backlink***
>
> Unknown interpreted text role "ref".

Handling OS and distro differences

Group variables files and the `group_by` module work together to help Ansible execute across a range of operating systems and distributions that require different settings, packages, and tools. The `group_by` module creates a dynamic group of hosts matching certain criteria. This group does not need to be defined in the inventory file. This approach lets you execute different tasks on different operating systems or distributions. For example:

```
---

- name: talk to all hosts just so we can learn about them
  hosts: all
  tasks:
    - name: Classify hosts depending on their OS distribution
      group_by:
        key: os_{{ ansible_facts['distribution'] }}

# now just on the CentOS hosts...

- hosts: os_CentOS
  gather_facts: False
  tasks:
    - # tasks that only happen on CentOS go in this play
```

The first play categorizes all systems into dynamic groups based on the operating system name. Later plays can use these groups as patterns on the `hosts` line. You can also add group-specific settings in group vars files. All three names must match: the name created by the `group_by` task, the name of the pattern in subsequent plays, and the name of the group vars file. For example:

```
---
# file: group_vars/all
asdf: 10

---
# file: group_vars/os_CentOS.yml
asdf: 42
```

In this example, CentOS machines get the value of '42' for asdf, but other machines get '10'. This can be used not only to set variables, but also to apply certain roles to only certain systems.

You can use the same setup with `include_vars` when you only need OS-specific variables, not tasks:

```
- hosts: all
  tasks:
```

```
  - name: Set OS distribution dependent variables
    include_vars: "os_{{ ansible_facts['distribution'] }}.yml"
  - debug:
      var: asdf
```

This pulls in variables from the group_vars/os_CentOS.yml file.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\ansible-devel\docs\docsite\rst\user_guide\[ansible-devel][docs][docsite][rst][user_guide]playbooks_best_practices.rst, line 166)

Unknown directive type "seealso".

```
.. seealso::

   :ref:`yaml_syntax`
       Learn about YAML syntax
   :ref:`working_with_playbooks`
       Review the basic playbook features
   :ref:`list_of_collections`
       Browse existing collections, modules, and plugins
   :ref:`developing_modules`
       Learn how to extend Ansible by writing your own modules
   :ref:`intro_patterns`
       Learn about how to select hosts
   `GitHub examples directory <https://github.com/ansible/ansible-examples>`_
       Complete playbook files from the github project source
   `Mailing List <https://groups.google.com/group/ansible-project>`_
       Questions? Help? Ideas?  Stop by the list on Google Groups
```