Tips Tricks

- General tips
- Ansible tips and tricks
- Sample Ansible setup

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 with meaningful comments 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 `callback_plugins`.

Avoid configuration-dependent content

To ensure that your automation project is easy to understand, modify, and share with others, you should avoid configuration-dependent content. For example, rather than referencing an `ansible.cfg` as the root of a project, you can use magic variables such as `playbook_dir` or `role_name` to determine paths relative to known locations within your project directory. This can

help to keep automation content flexible, reusable, and easy to maintain. For more information, see `special variables<special_variables>`.

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 plays, tasks, and blocks

Play, task, and block `- name:`'s are optional, but extremely useful. In its output, Ansible shows you the name of each named entity it runs. Choose names that describe what each play, task, and block 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 states, 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 the future) understand what a play or task (or variable setting) does, how it does it, and why.

Use fully qualified collection names

Use `fully qualified collection names (FQCN) <Fully Qualified Collection Name (FQCN)>` to avoid ambiguity in which collection to search for the correct module or plugin for each task.

For `builtin modules and plugins <ansible.builtin>`, use the `ansible.builtin` collection name as a prefix, for example, `ansible.builtin.copy`.

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 `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.

Group inventory by function

A system can be in multiple groups. See `intro_inventory` and `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 `playbooks_reuse_roles`.

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.

Use Execution Environments

Reduce complexity with portable container images known as `Execution Environments<getting_started_ee_index>`.

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 environments. You can also check for any syntax errors in the staging environment using the flag `--syntax-check` such as in the following example:

```
ansible-playbook --syntax-check
```

Update in batches

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

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 that match 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, the following play categorizes all systems into dynamic groups based on the operating system name:

yaml/tip_group_by.yaml

Subsequent plays can use these groups as patterns on the `hosts` line as follows:

yaml/tip_group_hosts.yaml

You can also add group-specific settings in group vars files. In the following example, CentOS machines get the value of '42' for asdf but other machines get '10'. You can also use group vars files to apply roles to systems as well as set variables.

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


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

> 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.

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

yaml/tip_include_vars.yaml

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

`yaml_syntax`
Learn about YAML syntax

`working_with_playbooks`
Review the basic playbook features

`list_of_collections`
Browse existing collections, modules, and plugins

`developing_modules`
Learn how to extend Ansible by writing your own modules

`intro_patterns`
Learn about how to select hosts

`Communication<communication>`
Got questions? Need help? Want to share your ideas? Visit the Ansible communication guide

Ansible tips and tricks

> **Making Open Source More Inclusive**
>
> Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. We ask that you open an issue or pull request if you come upon a term that we have missed. For more details, see our CTO Chris Wright's message.

Welcome to the Ansible tips and tricks guide. 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 the most sense for your organization and your goals.

ansible_tips_tricks sample_setup

Sample Ansible setup

You have learned about playbooks, inventory, roles, and variables. This section combines all those elements and outlines a sample setup for automating a web service.

The sample setup organizes playbooks, roles, inventory, and files with variables by function. Tags at the play and task level provide greater granularity and control. This is a powerful and flexible approach, but there are other ways to organize Ansible content. Your usage of Ansible should fit your needs, so feel free to modify this approach and organize your content accordingly.

Sample directory layout

This layout organizes most tasks in roles, with a single inventory file for each environment and a few playbooks in the top-level directory:

```
production                # inventory file for production servers
staging                   # inventory file for staging environment

group_vars/
   group1.yml             # here we assign variables to particular groups
   group2.yml
host_vars/
   hostname1.yml          # here we assign variables to particular systems
   hostname2.yml

library/                  # if any custom modules, put them here (optional)
module_utils/             # if any custom module_utils to support modules, put them here
(optional)
filter_plugins/           # if any custom filter plugins, put them here (optional)

site.yml                  # main playbook
webservers.yml            # playbook for webserver tier
dbservers.yml             # playbook for dbserver tier
tasks/                    # task files included from playbooks
    webservers-extra.yml  # <-- avoids confusing playbook with task files
```

```
roles/
    common/               # this hierarchy represents a "role"
        tasks/            #
            main.yml      #  <-- tasks file can include smaller files if warranted
        handlers/         #
            main.yml      #  <-- handlers file
        templates/        #  <-- files for use with the template resource
            ntp.conf.j2   #  <------- templates end in .j2
        files/            #
            bar.txt       #  <-- files for use with the copy resource
            foo.sh        #  <-- script files for use with the script resource
        vars/             #
            main.yml      #  <-- variables associated with this role
        defaults/         #
            main.yml      #  <-- default lower priority variables for this role
        meta/             #
            main.yml      #  <-- role dependencies and optional Galaxy info
        library/          # roles can also include custom modules
        module_utils/     # roles can also include custom module_utils
        lookup_plugins/   # or other types of plugins, like lookup in this case

    webtier/              # same kind of structure as "common" was above, done for the webtier
role
    monitoring/           # ""
    fooapp/               # ""
```

> By default, Ansible assumes your playbooks are stored in one directory with roles stored in a sub-directory called `roles/`. With more tasks to automate, you can consider moving your playbooks into a sub-directory called `playbooks/`. If you do this, you must configure the path to your `roles/` directory using the `roles_path` setting in the `ansible.cfg` file.

Alternative directory layout

You can also put each inventory file with its `group_vars`/`host_vars` in a separate directory. This is particularly useful if your `group_vars`/`host_vars` do not have that much in common in different environments. The layout could look like this example:

```
inventories/
   production/
      hosts               # inventory file for production servers
      group_vars/
         group1.yml       # here we assign variables to particular groups
         group2.yml
      host_vars/
         hostname1.yml    # here we assign variables to particular systems
         hostname2.yml

   staging/
      hosts               # inventory file for staging environment
      group_vars/
         group1.yml       # here we assign variables to particular groups
         group2.yml
      host_vars/
         stagehost1.yml   # here we assign variables to particular systems
         stagehost2.yml

library/
module_utils/
filter_plugins/

site.yml
webservers.yml
dbservers.yml

roles/
    common/
    webtier/
    monitoring/
    fooapp/
```

This layout gives you more flexibility for larger environments, as well as a total separation of inventory variables between different environments. However, this approach is harder to maintain, because there are more files. For more information on organizing group and host variables, see `splitting_out_vars`.

Sample group and host variables

These sample group and host files with variables contain the values that apply to each machine or a group of machines. For example, the data center in Atlanta has its own NTP servers. As a result, when setting up the `ntp.conf` file, you could use similar code as in this example:

```
---
# file: group_vars/atlanta
ntp: ntp-atlanta.example.com
backup: backup-atlanta.example.com
```

Similarly, hosts in the webservers group have some configuration that does not apply to the database servers:

```
---
# file: group_vars/webservers
apacheMaxRequestsPerChild: 3000
apacheMaxClients: 900
```

Default values, or values that are universally true, belong in a file called `group_vars/all`:

```
---
# file: group_vars/all
ntp: ntp-boston.example.com
backup: backup-boston.example.com
```

If necessary, you can define specific hardware variance in systems in the `host_vars` directory:

```
---
# file: host_vars/db-bos-1.example.com
foo_agent_port: 86
bar_agent_port: 99
```

If you use `dynamic inventory <dynamic_inventory>`, Ansible creates many dynamic groups automatically. As a result, a tag like `class:webserver` will load in variables from the file `group_vars/ec2_tag_class_webserver` automatically.

> You can access host variables with a special variable called `hostvars`. See `special_variables` for a list of these variables. The `hostvars` variable can access only host-specific variables, not group variables.

Sample playbooks organized by function

With this setup, a single playbook can define the entire infrastructure. The `site.yml` playbook imports two other playbooks. One for the webservers and one for the database servers:

> ```
> ---
> # file: site.yml
> - import_playbook: webservers.yml
> - import_playbook: dbservers.yml
> ```

The `webservers.yml` playbook, also at the top level, maps the configuration of the webservers group to the roles related to the webservers group:

> ```
> ---
> # file: webservers.yml
> - hosts: webservers
>   roles:
>     - common
>     - webtier
> ```

With this setup, you can configure your entire infrastructure by running `site.yml`. Alternatively, to configure just a portion of your infrastructure, run `webservers.yml`. This is similar to the Ansible `--limit` parameter but a little more explicit:

```
ansible-playbook site.yml --limit webservers
ansible-playbook webservers.yml
```

Sample task and handler files in a function-based role

Ansible loads any file called `main.yml` in a role sub-directory. This sample `tasks/main.yml` file configures NTP:

```
---
# file: roles/common/tasks/main.yml

- name: be sure ntp is installed
  yum:
    name: ntp
    state: present
  tags: ntp

- name: be sure ntp is configured
  template:
    src: ntp.conf.j2
    dest: /etc/ntp.conf
  notify:
    - restart ntpd
  tags: ntp

- name: be sure ntpd is running and enabled
  ansible.builtin.service:
    name: ntpd
    state: started
    enabled: true
  tags: ntp
```

Here is an example handlers file. Handlers are only triggered when certain tasks report changes. Handlers run at the end of each play:

```
---
# file: roles/common/handlers/main.yml
- name: restart ntpd
  ansible.builtin.service:
    name: ntpd
    state: restarted
```

See `playbooks_reuse_roles` for more information.

What the sample setup enables

The basic organizational structure described above enables a lot of different automation options. To reconfigure your entire infrastructure:

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

To reconfigure NTP on everything:

```
ansible-playbook -i production site.yml --tags ntp
```

To reconfigure only the webservers:

```
ansible-playbook -i production webservers.yml
```

To reconfigure only the webservers in Boston:

```
ansible-playbook -i production webservers.yml --limit boston
```

To reconfigure only the first 10 webservers in Boston, and then the next 10:

```
ansible-playbook -i production webservers.yml --limit boston[0:9]
ansible-playbook -i production webservers.yml --limit boston[10:19]
```

The sample setup also supports basic ad hoc commands:

```
ansible boston -i production -m ping
ansible boston -i production -m command -a '/sbin/reboot'
```

To discover what tasks would run or what hostnames would be affected by a particular Ansible command:

```
# confirm what task names would be run if I ran this command and said "just ntp tasks"
ansible-playbook -i production webservers.yml --tags ntp --list-tasks

# confirm what hostnames might be communicated with if I said "limit to boston"
ansible-playbook -i production webservers.yml --limit boston --list-hosts
```

Organizing for deployment or configuration

The sample setup illustrates a typical configuration topology. When you do multi-tier deployments, you will likely need some additional playbooks that hop between tiers to roll out an application. In this case, you can augment `site.yml` with playbooks like `deploy_exampledotcom.yml`. However, the general concepts still apply. With Ansible you can deploy and configure using the same utility. Therefore, you will probably reuse groups and keep the OS configuration in separate playbooks or roles from the application deployment.

Consider "playbooks" as a sports metaphor -- you can have one set of plays to use against all your infrastructure. Then you have situational plays that you use at different times and for different purposes.

Using local Ansible modules

If a playbook has a `./library` directory relative to its YAML file, you can use this directory to add Ansible modules automatically to the module path. This organizes modules with playbooks. For example, see the directory structure at the start of this section.

`yaml_syntax`
Learn about YAML syntax

`working_with_playbooks`
Review the basic playbook features

`list_of_collections`
Browse existing collections, modules, and plugins

`developing_modules`
Learn how to extend Ansible by writing your own modules

`intro_patterns`
Learn about how to select hosts

`Communication<communication>`
Got questions? Need help? Want to share your ideas? Visit the Ansible communication guide