Sample Ansible setup

- Sample directory layout
- Alternative directory layout
- Sample group and host variables
- Sample playbooks organized by function
- Sample task and handler files in a function-based role
- What the sample setup enables
- Organizing for deployment or configuration
- Using local Ansible modules

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

---

□□□ #2
□ □□□ □□□ 2026-10-31 14:45:29 UTC
□ □□□ □□□ 2026-10-31 14:45:31 UTC