docs.ansible.com/ansible/2.4/playbooks_best_practices.html

Here are some tips for making the most of Ansible and Ansible playbooks.

You can find some example playbooks illustrating these best practices in our ansible-examples repository. (NOTE: These may not use all of the features in the latest release, but are still an excellent reference!).

The following section shows one of many possible ways to organize playbook content.

Your usage of Ansible should fit your needs, however, not ours, so feel free to modify this approach and organize as you see fit.

One crucial way to organize your playbook content is Ansible's "roles" organization feature, which is documented as part of the main playbooks page. You should take the time to read and understand the roles documentation which is available here: Roles.

The top level of the directory would contain files and directories like so:

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

group_vars/
   group1                 # here we assign variables to particular groups
   group2                 # ""
host_vars/
   hostname1              # if systems need specific variables, put them
here
   hostname2              # ""

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                  # master playbook
webservers.yml            # playbook for webserver tier
dbservers.yml             # playbook for dbserver tier

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
        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/               # ""
```

Alternatively you can put each inventory file with its `group_vars` / `host_vars` in a separate directory. This is particularly useful if your `group_vars` / `host_vars` don't have that much in common in different environments. The layout could look something like this:

```
inventories/
   production/
      hosts               # inventory file for production servers
      group_vars/
         group1           # here we assign variables to particular groups
         group2           # ""
      host_vars/
         hostname1        # if systems need specific variables, put them
here
         hostname2        # ""

   staging/
      hosts               # inventory file for staging environment
      group_vars/
         group1           # here we assign variables to particular groups
         group2           # ""
      host_vars/
         stagehost1       # if systems need specific variables, put them
here
         stagehost2       # ""

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. The downside is that it is harder to maintain, because there are more files.

If you are using a cloud provider, you should not be managing your inventory in a static file. See Dynamic Inventory.

This does not just apply to clouds – If you have another system maintaining a canonical list of systems in your infrastructure, usage of dynamic inventory is a great idea in general.

If managing static inventory, it is frequently asked how to differentiate different types of environments. The following example shows a good way to do this. Similar methods of grouping could be adapted to dynamic inventory (for instance, consider applying the AWS tag "environment:production", and you'll get a group of systems automatically discovered named "ec2_tag_environment_production".

Let's show a static inventory example though. Below, the *production* file contains the inventory of all of your production hosts.

It is suggested that you define groups based on purpose of the host (roles) and also geography or datacenter location (if applicable):

```
# file: production

[atlanta-webservers]
www-atl-1.example.com
www-atl-2.example.com

[boston-webservers]
www-bos-1.example.com
www-bos-2.example.com

[atlanta-dbservers]
db-atl-1.example.com
db-atl-2.example.com

[boston-dbservers]
db-bos-1.example.com

# webservers in all geos
[webservers:children]
atlanta-webservers
boston-webservers

# dbservers in all geos
[dbservers:children]
atlanta-dbservers
boston-dbservers

# everything in the atlanta geo
[atlanta:children]
atlanta-webservers
atlanta-dbservers

# everything in the boston geo
[boston:children]
boston-webservers
boston-dbservers
```

This section extends on the previous example.

Groups are nice for organization, but that's not all groups are good for. You can also assign variables to them! For instance, atlanta has its own NTP servers, so when setting up ntp.conf, we should use them. Let's set those now:

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

Variables aren't just for geographic information either! Maybe the webservers have some configuration that doesn't make sense for the database servers:

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

If we had any default values, or values that were universally true, we would put them in a file called group_vars/all:

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

We can define specific hardware variance in systems in a host_vars file, but avoid doing this unless you need to:

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

Again, if we are using dynamic inventory sources, many dynamic groups are automatically created. So a tag like "class:webserver" would load in variables from the file "group_vars/ec2_tag_class_webserver" automatically.

In site.yml, we import a playbook that defines our entire infrastructure. This is a very short example, because it's just importing some other playbooks:

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

In a file like webservers.yml (also at the top level), we map the configuration of the webservers group to the roles performed by the webservers group:

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

The idea here is that we can choose to configure our whole infrastructure by "running" site.yml or we could just choose to run a subset by running webservers.yml. This is analogous to the "–limit" parameter to ansible but a little more explicit:

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

Below is an example tasks file that explains how a role works. Our common role here just sets up NTP, but it could do more if we wanted:

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

- name: be sure ntp is installed
  : name=ntp state=installed
  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
  service: name=ntpd state=started enabled=yes
  tags: ntp
```

Here is an example handlers file. As a review, handlers are only fired when certain tasks report changes, and are run at the end of each play:

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

See Roles for more information.

Above we've shared our basic organizational structure.

Now what sort of use cases does this layout enable? Lots! If I want to reconfigure my whole infrastructure, it's just:

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

What about just reconfiguring NTP on everything? Easy.:

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

What about just reconfiguring my webservers?:

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

What about just my webservers in Boston?:

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

What about just the first 10, and then the next 10?:

```
ansible-playbook -i production webservers.yml --limit boston[1:10]
ansible-playbook -i production webservers.yml --limit boston[11:20]
```

And of course just basic ad-hoc stuff is also possible.:

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

And there are some useful commands to know (at least in 1.1 and higher):

```
# 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
```

The above setup models a typical configuration topology. When doing multi-tier deployments, there are going to be some additional playbooks that hop between tiers to roll out an application. In this case, 'site.yml' may be augmented by playbooks like 'deploy_exampledotcom.yml' but the general concepts can still apply.

Consider "playbooks" as a sports metaphor – you don't have to just have one set of plays to use against your infrastructure all the time – you can have situational plays that you use at different times and for different purposes.

Ansible allows you to deploy and configure using the same tool, so you would likely reuse groups and just keep the OS configuration in separate playbooks from the app deployment.

As also mentioned above, a good way to keep your staging (or testing) and production environments separate is to use a separate inventory file for staging and production. This way you pick with -i what you are targeting. Keeping them all in one file can lead to surprises!

Testing things in a staging environment before trying 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.

Understand the 'serial' keyword. If updating a webserver farm you really want to use it to control how many machines you are updating at once in the batch.

See Delegation, Rolling Updates, and Local Actions.

The 'state' parameter is optional to a lot of modules. Whether 'state=present' or 'state=absent', it's always best to leave that parameter in your playbooks to make it clear, especially as some modules support additional states.

We're somewhat repeating ourselves with this tip, but it's worth repeating. A system can be in multiple groups. See Inventory and Patterns. Having groups named after things like *webservers* and *dbservers* is repeated in the

examples because it's a very powerful concept.

This allows playbooks to target machines based on role, as well as to assign role specific variables using the group variable system.

See Roles.

When dealing with a parameter that is different between two different operating systems, a great way to handle this is by using the group_by module.

This makes a dynamic group of hosts matching certain criteria, even if that group is not defined in the inventory file:

```
---

# talk to all hosts just so we can learn about them
- hosts: all
  tasks:
     - group_by: key=os_{{ ansible_distribution }}

# now just on the CentOS hosts...

- hosts: os_CentOS
  gather_facts: False
  tasks:
     - # tasks that only happen on CentOS go here
```

This will throw all systems into a dynamic group based on the operating system name.

If group-specific settings are needed, this can also be done. For example:

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

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

In the above 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.

Alternatively, if only variables are needed:

```
- hosts: all
  tasks:
    - include_vars: "os_{{ ansible_distribution }}.yml"
    - debug: var=asdf
```

This will pull in variables based on the OS name.

If a playbook has a `./library` directory relative to its YAML file, this directory can be used to add ansible modules that will automatically be in the ansible module path. This is a great way to keep modules that go with a playbook together. This is shown in the directory structure example at the start of this section.

It is possible to leave off the 'name' for a given task, though it is recommended to provide a description about why something is being done instead. This name is shown when the playbook is run.

When you can do something simply, do something simply. Do not reach to use every feature of Ansible together, all at once. Use what works for you. 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, and may be a good opportunity to simplify things.

Use version control. Keep your playbooks and inventory file in git (or another version control system), and commit when you make changes to them. This way you have an audit trail describing when and why you changed the rules that are automating your infrastructure.

For general maintenance, it is often easier to use `grep`, or similar tools, to find variables in your Ansible setup. Since vaults obscure these variables, it is best to work with a layer of indirection. When running a playbook, Ansible finds the variables in the unencrypted file and all sensitive variables come from the encrypted file.

A best practice approach for this is to start with a `group_vars/` subdirectory named after the group. Inside of this subdirectory, create two files named `vars` and `vault`. Inside of the `vars` file, define all of the variables needed, including any sensitive ones. Next, copy all of the sensitive variables over to the `vault` file and prefix these variables with `vault_`. You should adjust the variables in the `vars` file to point to the matching `vault_` variables using jinja2 syntax, and ensure that the `vault` file is vault encrypted.

This best practice has no limit on the amount of variable and vault files or their names.

See also