Ansible Best Practices

Optimize playbooks content for readability

Reduce complexity, aim at simplification, but yet try to use as many variables and templates as it is required to keep your code flexible and reusable by others.

Name your plays and tasks

Always name your plays and tasks. Never leave tasks without names assigned. Otherwise, nobody understands what the task may be doing and that is the biggest driver of user anxiety.

```
---
# tasks file for myrole

# Wrong:
- systemd:
    name: ntpd
    state: started
    enabled: true

# Perfect:
- name: start and enable NTP
  systemd:
    name: ntpd
    state: started
    enabled: true
  tags:
    - myrole_ntp
```

Use prefixes with Host Group names

Use meaningful group names that communicate their purpose and/or technology. Prefixes are a perfect solution to achieve this, f.g. `pg-dev` (postgres DEV), `ora-uat` (oracle UAT).

```
[pg-dev]
pgserver01
```

Use role names as prefixes for variables

Prefixing variables is particularly vital with developing reusable and portable roles. Use role names as prefixes for variables. Since role names are in a common namespace, prefixing format described here addresses any concerns related to a potential variable names collision between different roles used in a play.

```
---
# defaults file for myrole
```

```
# disable/enable ntp service
myrole_configure_ntp: true
```

Use role names as prefixes for tags

The same rule described above is applicable for tags. There is no exception here.

```
---
# tasks file for myrole

- name: start and enable NTP
  systemd:
    name: ntpd
    state: started
    enabled: true
  tags:
    - myrole_services
    - myrole_ntp
```

Use special prefixes for registered variables

Use `_rolename` prefix (i.e. '_' underscore and a role name) when registering variables. It makes much easier to debug any problem and, again, limits the risk of variable collisions.
?

```
---
# tasks file for myrole

- name: a shell command
  shell: |
    /usr/bin/foo
  register: _myrole_foo_result
  tags:
    - myrole_foo_output
```

Use Native YAML Syntax over key=value pairs shorthand

Therefore, instead:

```
- name: create a directory
  file: path=/tmp/some_directory state=directory mode=0755
```

use native YAML syntax

```
- name: create a directory
  file:
    path: /tmp/some_directory
    state: directory
    mode: 0755
```

That style of formatting increases readability, reduces horizontal scrolling, and the task parameters are easily distinguished from the next.

Use comments and document your variables

Generous use of comments (which start with '#'), is encouraged. They are mandatory for a role default variables (`role/defaults`).

```
---
# Example:

# nr of hugepages for a host
postgresql_host_nr_hugepages: 100

# configure sudoers entries (this requires '#includedir /etc/sudoers.d' in /etc/sudoers)
postgresql_host_configure_sudoers: true

# VG used to store postgres data files
postgresql_host_datavg: datavg
```

Use Vaults to hide sensitive data

Sometimes it is required to include sensitive data (i.e. secrets) in variables. Vaults obscure these variables so use this layer of indirection, encrypt variables and never expose secrets to others. That is especially important when keeping your code in a remote git repository (Bitbucket/Github), where exposing sensitive data to others may be a serious security breach.

```
---
#------------------------------------------------
# Example of secrets (encrypted variables)
#------------------------------------------------
my_secret_nagios_password: !vault |
          $ANSIBLE_VAULT;1.1;AES256
          376138373662633863643832393236333666233623331383531663966303138353564376238613 6
          396138363433363937613730346231366166346436353637 0a3636666330623936343563636636 6 3
          643563326264303661316230333237393637316461356539313931666630616634633439326636 3 8
          383931353937656336 0a396262623366623734633030663034333531653465636466363934653032
          3134
```

Create group_vars subdirectories named after the groups

Avoid `group_vars/group_name` files, use `group_vars/group_name` subdirectories instead. Inside of this subdirectory, create files named `vars` (for regular variables) and `secrets` (for encrypted variables). This configuration is more flexible over `group_vars/group_name` files.

```
group_vars/
   group1/
      vars                # here we assign variables to particular groups
      secrets             # here we assign secrets to particular groups
   group2/
      vars                # here we assign variables to particular groups
      secrets             # here we assign secrets to particular groups
```

Create host_vars subdirectories named after the hosts

The same rule described above for `group_vars` is applicable for `host_vars`.

```
host_vars/
   hostname1/
      vars                # here we assign variables to particular systems
      secrets             # here we assign secrets to particular systems
   hostname2/
      vars                # here we assign variables to particular systems
      secrets             # here we assign secrets to particular systems
```

Use Group And Host Variables

Assign variables to particular groups or systems to override role defaults. Keep `role_name/defaults` system-agnostic, i.e. not specific to a customer environment, use placeholders. Group and Host variables are meant to define a particular system and set up variable values applicable for a customer.

For example, this is an excerpt from oracle-instantclient role defaults definition:

```
$ cat oracle-instantclient/defaults/main.yml
```

```
---
# sqlnet.ora lines
oracle_instantclient_sqlnet_ora:
  "name.default_zone": 'emea.foo.com'    # Note placeholders
  "names.default_domain": 'emea.foo.com'  # Note placeholders
```

Excerpt from group_vars/group1/vars file:

```
---
#-----------------------------------------------------
# overrides role 'oracle-instantclient' variables
#-----------------------------------------------------

# sqlnet.ora lines
oracle_instantclient_sqlnet_ora:
  "name.default_zone": 'emea.realcompany.com'
  "names.default_domain": 'emea.realcompany.com'
```

Make sure your playbook successfully runs in check mode

When ansible-playbook is executed with `--check` it will not make any changes on remote systems.

Other users would be happy to test your code in a safe way before running it on a live system. Therefore, launch your playbook in check mode (dry run) and make sure this simulation executes successfully and check mode is handled properly.

```
ansible-playbook foo.yml --check
```

Always create README.md files

Create README.md files for your roles and playbooks. It is mandatory. Read about why you should make it and introduction to basic Markdown syntax.

A playbook README.md file should contain at least the following sections:

- short description
- supported OS
- requirements
- examples of most common use cases along with `--tags`, `--skip-tags` and/or `--extra-vars`
- author information

On the other hand, include the following sections to a role README.md:

- detailed description
- supported OS
- requirements & dependencies
- list of variables from `role/defaults` along with comments and description
- example of a playbook which calls the role
- default variables overrode examples

Use label directive for loops

When using complex data structures for looping the display might get indecipherable, hard to debug and understand. This is where label directive may help. This will display just the 'label' field instead of the whole structure per 'item'.

```
---
# Example:
  - name: create users
    user:
      name: "{{ item.username }}"
      state: present
    with_items:
      - username: bjones
        first_name: bob
        last_name: jones
        home_dir: /users/bjones
        password: secret1
      - username: acook
        first_name: anne
        last_name: cook
        home_dir: /users/acook
        password: secret2
    loop_control:
      label: "[user {{ item.username }}; home_dir: {{ item.home_dir }}]"
    tags:
      - create_users
```

This will produce:

```
TASK [create users]
*********************************************************************************
*********************************************
changed: [app1] => (item=[user bjones; home_dir: /users/bjones])
changed: [app2] => (item=[user bjones; home_dir: /users/bjones])
changed: [app1] => (item=[user acook; home_dir: /users/acook])
changed: [app2] => (item=[user acook; home_dir: /users/acook])
```

instead of:

```
TASK [create users]
*********************************************************************************
*********************************************
changed: [app1] => (item={u'username': u'bjones', u'first_name': u'bob',
u'password': u'secret1', u'home_dir': u'/users/bjones', u'last_name':
u'jones'})
changed: [app2] => (item={u'username': u'bjones', u'first_name': u'bob',
u'password': u'secret1', u'home_dir': u'/users/bjones', u'last_name':
u'jones'})
changed: [app1] => (item={u'username': u'acook', u'first_name': u'anne',
u'password': u'secret2', u'home_dir': u'/users/acook', u'last_name': u'cook'})
changed: [app2] => (item={u'username': u'acook', u'first_name': u'anne',
u'password': u'secret2', u'home_dir': u'/users/acook', u'last_name': u'cook'})
```

Use asserts to validate the parameters

You can check the parameters before using them and avoid many pitfalls. Not running things that will break down is a good thing to do. Write meaningful error messages.

```
---
- name: "Validate version is a number, > 0"
  assert:
    that:
      - "{{ version | int }} > 0"
    msg: "'version' must be a number and > 0"
```