Ansible Best Practices

Ansible v3.5.0

Table of Contents

Summary

This guide is a summary of best practices for automating tasks utilizing Ansible playbooks with the assumption that the end user has limited Ansible experience. This guide will outline best practices of the overall ansible tower organization as well as introduce the development strategy “Make it work, Make it right, Make it fast” to help the developer remain focused on getting the work done, and avoid falling into the trap of premature optimization while creating.

This guide is intended to cover what is widely accepted as the most critical best practices, however Ansible has hundreds, if not thousands, of options and features. Included in the Appendix is a list links for further information and education on Ansible and Ansible tower.

Playbook Development Strategy

Ansible philosophy

When developing automated solutions with ansible one should understand Ansible's philosophy. All best practices relate back to this philosophy in one way or another.

- Complexity kills productivity
    - Ansible strives to reduce complexity in how they designed Ansible tools and encourage you to do the same. Strive for simplification in what you automate.
- Optimize your Ansible content for readability
    - If done properly, it can be the documentation of your workflow automation.
- Think declaratively
    - Ansible is a desired state engine by design. If you're trying to "write code" in your plays and roles, you're setting yourself up for failure. Ansible's YAML-based playbooks were never meant to be for programming.

Although, as stated above, Ansible is not meant for programing, its YAML-based playbooks must be "developed". We recommend the following development strategy in developing playbooks.

Make it work:

The first step is to make it work, have the playbook do what it is supposed to do, even if the code is not elegant. Just make sure the playbook works and works repeatedly. If you cannot make "it" work the "it" may be too big or complex. Review and try to reduce the scope of what "it" is. Then move on to steps two and three before continuing with the larger scoped work.

In this phase you will apply the following best practices:

- Static Inventory files
- Vaulted Configuration files
- Tasks
- Plays
- Name your plays and tasks
- Version Control
- Use Native YAML Syntax

Make it Right:

Once you have a working solution it is time to review your playbook and identify areas of improvement in your code. This is the time to identify any potential issues by testing multiple scenarios and to ensure your code is reusable for other similar tasks.

In this phase you will apply the following best practices:

- Dynamic Inventory files
- Roles
- Variables
    - Special variables
- Handlers
- Vault

Make it Fast:

If the playbook is not fast (in terms of performance) enough already, now is the time to measure and tune the playbooks performance. Performance tuning in Ansible could be as simple as increasing the number of parallel jobs to more complex analysis of what checks and validation steps are being performed and disabling some of them. If the Ansible script is calling another application/script to perform a task, that artifact may also need to be evaluated

In this phase you will apply the following best practices:

- Disabling Gather_facts
- Changed When
- Cache_valid_Time

Ansible Best Practices

Use Prefixes and Human Meaningful Names with Variables

Ansible has a powerful variable processing system that collects metadata from various sources and manages their merge and context as a play runs on your hosts. A lot of effort goes into making that power as easy and transparent as possible to users.

Variable scoping and namespacing was intentionally limited and shallow by design to make it easier to understand and debug.

A limitation to this approach is that you run the risk of variable collisions if two different pieces of information, like a port number, are stored in the same variable name. (Having a web server listen to port 22 is not the sort of fun you want to have.)

As a best practice, we recommend prefixing variables with the source or target of the data it represents. Prefixing variables is particularly vital with developing reusable and portable roles.

```
apache_max_keepalive: 25
apache_port: 80
tomcat_port: 8080
```

You can also dramatically improve the readability of your plays for a bit of extra verbosity with using human-meaningful names that communicate their purpose and usage to others or even yourself at a later date.

Whitespace and Comments:

Generous use of whitespace to break things up and use of comments (which start with '#'), is encouraged to enhance the readability of the playbooks.

Use Modules Before Run Commands

Run commands are what we collectively call the `command`, `shell`, `raw` and `script` modules that enable users to do command line operations in different ways. They're a great catch all mechanism for getting things done, but they should be used sparingly and as a last resort. The reasons are many and varied. The overuse of run commands is often a symptom of TL; DR(Too long; didn't read) in Ansible and common amongst those just becoming familiar with Ansible for automating their work. They use `shell` to fire off a bash command they already know without stopping to look at the Ansible docs. That works well enough initially, but it undermines the value of automating with Ansible and sets things up for problems down the road.

The most important thing to consider is that these run commands have little logic to them and no concept of desired state like a typical Ansible module.

That `shell` that succeeded the first time you ran your play may fail the next time when something already exists. That's unless you `ignore_errors` on that task. But how do you catch a real error like wrong permissions? Now you have to register the result of that first command and follow it with another task that implements conditional logic to check if an error occurred in the first and handle it.

Use Ansible Galaxy to find and share roles

Ansible Galaxy is a hub for finding, reusing, and sharing Ansible content with the rest of the Ansible community. You download, install, create, and manage roles with the `ansible-galaxy` command line interface. The CLI ships as part of the regular Ansible package and should already be installed wherever you have Ansible.

Essential ansible-galaxy commands

- `ansible-galaxy init <role_name>`
  Use this command to generate the directory and file structure and to stub out the README and metadata files for new roles you want to create and share.

```
roles
└── my_role
    ├── README.md
    ├── defaults
    │   └── main.yml
    ├── handlers
    │   └── main.yml
    ├── meta
    │   └── main.yml
    ├── tasks
    │   └── main.yml
    ├── tests
    │   ├── inventory
    │   └── test.yml
    └── vars
        └── main.yml
```

  Ansible creates the files; you just have to customize them. It's always a best practice to make sure you include the relevant documentation needed for each directory.

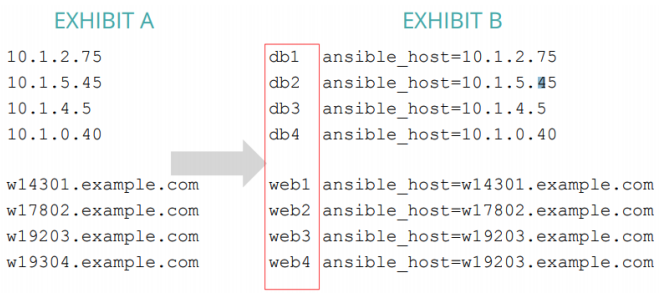
- `ansible-galaxy install`
  Use this command to download and install roles published to the Ansible Galaxy community. Someone may have already written the role you need. You might not have to reinvent the wheel.
- `ansible-galaxy import`
  Use this command to import roles from any repository to which you have access. Depending on how your teams write and share roles, this can open up some really cool possibilities. For example, roles can live in individual team or org repos but be imported and used from a central location.

Inventory files

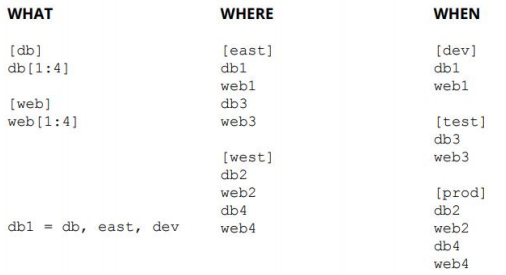
Static Inventory files

When initial starting automated task keep it simple, utilize static inventory files. And follow these best practices

- Give inventory nodes meaningful names



- Group hosts for easier inventory selection and less conditional tasks



Dynamic Inventory files

If possible, utilize a single source of truth for your inventory.
This will:

- Stay in sync automatically
- Reduce human error

Ansible provides modules for interfacing with most cloud providers as well as VMware
An example of the VMware Dynamic inventory is in the Appendix.

Tasks

A task is the most basic function of an Ansible play. Task are the main building block in Ansible, consisting of 2 main parts **module** and **arguments** for that module

Simple example:

```
ansible –m command –a “hostname” all
```

There are many modules available in ansible, refer to the documentation for more information

- Cloud modules
- Commands modules
- Database modules
- Files modules

Plays

Each playbook is an aggregation of one or more plays in it. Playbooks are structured using Plays. There can be more than one play inside a playbook.

For starters, here’s a playbook, `verify-apache.yml` that contains just one play:

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
```

```
remote_user: root
tasks:
- name: ensure apache is at the latest version
  yum:
    name: httpd
    state: latest
- name: write the apache config file
  template:
    src: /srv/httpd.j2
    dest: /etc/httpd.conf
  notify:
  - restart apache
- name: ensure apache is running
  service:
    name: httpd
    state: started
handlers:
  - name: restart apache
    service:
      name: httpd
      state: restarted
```

To run execute:

```
ansible-playbook verify-apache.yml
```

"Name" Your Plays and Tasks

Always `name` your plays and tasks. Adding name with a human meaningful description better communicates the intent to users when running a play. Consider this example play and its standard Ansible output.

YAML

```
- hosts: web
 tasks:
 - yum:
     name: httpd
     state: latest

 - service:
     name: httpd
     state: started
     enabled: yes
```

Output

```
PLAY [web] *********************************************************

TASK [setup] *******************************************************
ok: [web1]

TASK [yum] *********************************************************
ok: [web1]

TASK [service] *****************************************************
ok: [web1]
```

Without knowing what's in the play, another user can see the play is doing something with yum and something with service, but what exactly? They aren't even sure what the purpose of the play is except for maybe the filename or some external documentation source. Let's look at that same example with `name` declared on the play and all of its tasks.

YAML

```
- hosts: web
  name: installs and starts apache
  tasks:
    - name: install apache packages
      yum:
        name: httpd
        state: latest

    - name: starts apache service
      service:
        name: httpd
        state: started
        enabled: yes
```

Output

```
PLAY [install and starts apache] ***********************************

TASK [setup] *******************************************************
ok: [web1]

TASK [install apache packages] *************************************
ok: [web1]

TASK [starts apache service] ***************************************
ok: [web1]
```

Much better. It is now much more apparent what this playbook is doing right in the output of your playbook run. It also aids the usage of the `--list-tasks` switch in `ansible-playbook`.

Version Control

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.

Use Native YAML Syntax

At its core, the Ansible playbook runner is a YAML parser with added logic such as command line key=value pairs shorthand. While convenient when cranking out a quick playbook or a docs example, that style of formatting reduces readability. We recommend you refrain from using that shorthand (even with YAML folded style) as a best practice.

Here is an example of some tasks using the key=value shorthand:

YAML

```
- name: install telegraf
  yum:
    name: telegraf-{{ telegraf_version }} state=present update_cache=yes disable_gpg_check=yes
enablerepo=telegraf
  notify: restart telegraf

- name: configure telegraf
  template: src=telegraf.conf.j2 dest=/etc/telegraf/telegraf.conf
  notify: restart telegraf

- name: start telegraf
  service: name=telegraf state=started enabled=yes
```

Now here are the same tasks using native YAML syntax:

YAML

```yaml
- name: install telegraf
  yum: telegraf-{{ telegraf_version }}
    state: present
    update_cache: yes
    disable_gpg_check: yes
    enablerepo: telegraf
  notify: restart telegraf

- name: configure telegraf
  template:
    src: telegraf.conf.j2
    dest: /etc/telegraf/telegraf.conf
  notify: restart telegraf

- name: start telegraf
  service:
    name: telegraf
    state: started
    enabled: yes
```

Native YAML has more lines; however, those lines are shorter, reducing horizontal scrolling and line wrapping. It lets the eyes scan straight down the play. The task parameters are stacked and easily distinguished from the next. Native YAML syntax also has the benefit of improved syntax highlighting in virtually any modern text editor out there. Being native YAML, editors such as vim and Atom will highlight YAML keys (module names, directives, parameter names) from their values further aiding the readability of your content. Many of our own docs use this shorthand for legacy reasons though we're progressively changing that. (Documentation pull requests accepted.)

Use roles to group related tasks

Roles make Playbooks reusable in Ansible

While it is possible to write a playbook in one very large file (and you might start out writing playbooks this way), eventually you'll want to reuse files and start to organize things. In Ansible, there are three ways to do this: includes, imports, and roles.

- All `import*` statements are pre-processed at the time playbooks are parsed.
- All `include*` statements are processed as they are encountered during the execution of the playbook.

Roles are better way to modularize ansible playbooks than imports and includes

Roles expect files to be in certain directory names. Roles must include at least one of these directories, however it is perfectly fine to exclude any which are not being used. When in use, each directory must contain a `main.yml` file, which contains the relevant content:

- `tasks` - contains the main list of tasks to be executed by the role.
- `handlers` - contains handlers, which may be used by this role or even anywhere outside this role.
- `defaults` - default variables for the role
- `vars` - other variables for the role
- `files` - contains files which can be deployed via this role.

- `templates` - contains templates which can be deployed via this role.
- `meta` - defines some meta data for this role.

Ansible provides a way to scaffold the files for roles:

```
ansible-galaxy init <folder-name>
```

The classic (original) way to use roles is via the `roles:` option for a given play:

```
---
- hosts: webservers
  roles:
    - common
    - webservers
```

the order of execution for your playbook is as follows:

- Any `pre_tasks` defined in the play.
- Any handlers triggered so far will be run.
- Each role listed in `roles` will execute in turn. Any role dependencies defined in the roles `meta/main.yml` will be run first, subject to tag filtering and conditionals.
- Any `tasks` defined in the play.
- Any handlers triggered so far will be run.
- Any `post_tasks` defined in the play.
- Any handlers triggered so far will be run.

Variables

Adding Variables to Roles to make the Playbooks more reusable and efficient.
You can define variables directly in a playbook:

```
- hosts: webservers
  vars:
    http_port: 80
```

This can be nice as it's right there when you are reading the playbook.

Once you've defined variables, you can use them in your playbooks using the Jinja2 templating system. Here's a simple Jinja2 template:

```
My amp goes to {{ max_amp_value }}
```

There are other places where variables can come from, but these are a type of variable that are discovered, not set by the user.
Facts are information derived from speaking with your remote systems. You can find a complete set under the ansible_facts variable, most facts are also 'injected' as top level variables preserving the ansible_ prefix, but some are dropped due to conflicts. This can be disabled via the INJECT_FACTS_AS_VARS setting.

Special variables

Whether or not you define any variables, you can access information about your hosts with the Special Variables Ansible provides, including "magic" variables, facts, and connection

variables. Magic variable names are reserved - do not set variables with these names. The variable `environment` is also reserved.

The most commonly used magic variables are `hostvars`, `groups`, `group_names`, and `inventory_hostname`.

`hostvars` lets you access variables for another host, including facts that have been gathered about that host. You can access host variables at any point in a playbook. Even if you haven't connected to that host yet in any play in the playbook or set of playbooks, you can still get the variables, but you will not be able to see the facts.

If your database server wants to use the value of a 'fact' from another node, or an inventory variable assigned to another node, it's easy to do so within a template or even an action line:

```
{{ hostvars['test.example.com']['ansible_facts']['distribution'] }}
```

`groups` is a list of all the groups (and hosts) in the inventory. This can be used to enumerate all hosts within a group.
For example:

```
{% for host in groups['app_servers'] %}
  # something that applies to all app servers.
{% endfor %}
```

A frequently used idiom is walking a group to find all IP addresses in that group.

```
{% for host in groups['app_servers'] %}
  {{ hostvars[host]['ansible_facts']['eth0']['ipv4']['address'] }}
{% endfor %}
```

You can use this idiom to point a frontend proxy server to all of the app servers, to set up the correct firewall rules between servers, etc. You need to make sure that the facts of those hosts have been populated before though, for example by running a play against them if the facts have not been cached recently

`group_names` is a list (array) of all the groups the current host is in. This can be used in templates using Jinja2 syntax to make template source files that vary based on the group membership (or role) of the host:

```
{% if 'webserver' in group_names %}
  # some part of a configuration file that only applies to webservers
{% endif %}
```

`inventory_hostname` is the name of the hostname as configured in Ansible's inventory host file. This can be useful when you've disabled fact-gathering, or you don't want to rely on the discovered hostname `ansible_hostname`. If you have a long FQDN, you can use `inventory_hostname_short`, which contains the part up to the first period, without the rest of the domain.

Handlers

Handlers are only fired when certain tasks report changes, and are run at the end of each play:

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

When a task fails on a host, handlers which were previously notified will *not* be run on that host. This can lead to cases where an unrelated failure can leave a host in an unexpected state. For example, a task could update a configuration file and notify a handler to restart some service. If a task later on in the same play fails, the service will not be restarted despite the configuration change.

You can change this behavior with the `--force-handlers` command-line option, or by including `force_handlers: True` in a play, or `force_handlers = True` in ansible.cfg. When handlers are forced, they will run when notified even if a task fails on that host. (Note that certain errors could still prevent the handler from running, such as a host becoming unreachable.)

Controlling what defines failure:
Ansible lets you define what "failure" means in each task using the `failed_when` conditional. As with all conditionals in Ansible, lists of multiple `failed_when` conditions are joined with an implicit `and`, meaning the task only fails when *all* conditions are met. If you want to trigger a failure when any of the conditions is met, you must define the conditions in a string with an explicit `or` operator.

Vault

The "Vault" is a feature of Ansible that allows you to keep sensitive data such as passwords or keys protected at rest, rather than as plaintext in playbooks or roles. These vaults can then be distributed or placed in source control.

There are 2 types of vaulted content and each has their own uses and limitations:

- Vaulted files:
  - The full file is encrypted in the vault, this can contain Ansible variables or any other type of content.
  - It will always be decrypted when loaded or referenced, Ansible cannot know if it needs the content unless it decrypts it.
  - It can be used for inventory, anything that loads variables (i.e vars_files, group_vars, host_vars, include_vars, etc) and some actions that deal with files (i.e M(copy), M(assemble), M(script), etc).
- Single encrypted variable:
  - Only specific variables are encrypted inside a normal 'variable file'.
  - Does not work for other content, only variables.
  - Decrypted on demand, so you can have vaulted variables with different vault secrets and only provide those needed.
    - You can mix vaulted and non-vaulted variables in the same file, even inline in a play or role.

To enable this feature, a command line tool, ansible-vault is used to edit files, and a command line flag `--ask-vault-pass`, `--vault-password-file` or `--vault-id` is used

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.

```
ansible-playbook site.yml --ask-vault-pass
```

or

```
ansible-playbook site.yml --vault-password-file ~/.vault_pass.txt
```

Vaulted Configuration files

Configuration files often contains passwords in plain text, a security risk, encrypt your entire inventory configuration file to remediate this risk.

Encrypt a valid inventory configuration file as follows:

```
$ ansible-vault encrypt <filename>.vmware.yml
  New Vault password:
  Confirm New Vault password:
  Encryption successful
```

Utilize this vaulted inventory configuration file using:

```
$ ansible-inventory -i filename.vmware.yml --list --vault-password-file=/path/to/vault_password_file
```

Clean Up Your Debugging Messages

When developing Ansible content, it can be useful to drop in a `debug` task to display the content of a variable while your play runs. That's not so nice, even downright disconcerting, when your Playbook goes into production and some unwitting ops manager runs a play and sees your debugging messages on their screen.

That's why it is a best practice to clean up those debug statement in your Ansible content.

In the past, you had to delete or comment out your debug tasks. That was a suboptimal approach though.

Starting in version 2.1, the Ansible debug module supports a `verbosity` parameter that suppresses output unless the play is being run with a sufficiently high enough verbosity level.

YAML

```
- debug:
   msg: "This always displays"

- debug:
   msg: "This only displays with ansible-playbook -vv+"
   verbosity: 2
```

Now you users only see all of that debugging information unless they really want it.

Don't use ignore_errors

If Ansible encounters an error on a host when executing a task from a playbook, it won't continue running the playbook. Mostly this is a good thing, but sometimes certain errors shouldn't stop the

playbook from running. When this happens, it can be tempting to set `ignore_errors: yes` on the task and move on.

However, there are better, safer ways to handle errors. The `ignore_errors` setting swallows all errors, even ones you may not be expecting, and you risk leaving your host in a broken or unstable state. You can achieve less-brittle error handling and better control over how your tasks succeed or fail using the following Ansible built-ins:

- `failed_when:` Use this setting to specify exactly what constitutes failure, rather than just relying on the exit code. Maybe a command has really failed only when certain text is present in standard error (stderr), for example.
- `changed_when:` Use this setting to tell Ansible exactly when a task should register as "changed." This is especially important to use with modules like `shell` or `command`, since these will always run and will always report "changed" otherwise. This can also gives you better control over notifying handlers, since they're often notified by changes.
- `wait_for:` Use this setting to tell Ansible to wait until a specific condition is met before continuing (for example, wait for a port to become available on a service before configuring another service to talk to that port).
- `do-until` **loops**: You can set a module to retry until it registers a specified result, or to retry for a specified number of times with a specified delay between attempts. By default, Ansible sets `retries` at 3 and `delay` at 5.
- `block` and `rescue:` You can use a block section to set data or directives for a group of tasks (e.g., configuration settings), and then use a rescue section to recover from any error generated when the block section was executed. The ability to rescue errors from blocks gives you a ton of flexibility in how you handle errors; you can use them to `flush_handlers`, make callbacks, execute additional tasks, print debug messages, and much more.

Disabling gather_facts:

If you know you don't need any fact data about your hosts, and know everything about your systems centrally, you can turn off fact gathering. This has advantages in scaling Ansible in push mode with very large numbers of systems, mainly, or if you are using Ansible on experimental platforms. In any play, just do this:

```
- hosts: whatever
  gather_facts: no
```

changed_when:

When a shell/command or other module runs it will typically report "changed" status based on whether it thinks it affected machine state.
Sometimes you will know, based on the return code or output that it did not make any changes, and wish to override the "changed" result such that it does not appear in report output or does not cause handlers to fire:

```
tasks:

 - shell: /usr/bin/billybass --mode="take me to the river"
   register: bass_result
   changed_when: "bass_result.rc != 2"
```

```
# this will never report 'changed' status
- shell: wall 'beep'
  changed_when: False
```

You can also combine multiple conditions to override “changed” result:

```
- command: /bin/fake_command
  register: result
  ignore_errors: True
  changed_when:
    - '"ERROR" in result.stderr'
    - result.rc == 2
```

Appendix

VMware Dynamic Inventory Plugin

The VMware dynamic inventory plugin dynamically queries VMware APIs and tells Ansible what nodes can be managed.

To use the VMware dynamic inventory plugin, you need to enable it first by specifying the following in the `ansible.cfg` file:

```
[inventory]
enable_plugins = vmware_vm_inventory
```

Then, create a file that ends in `.vmware.yml` or `.vmware.yaml` in your working directory.
The `vmware_vm_inventory` script takes in the same authentication information as any VMware module.

Example of a valid inventory file:

```
plugin: vmware_vm_inventory
strict: False
hostname: 10.65.223.31
username: administrator@vsphere.local
password: Esxi@123$%
validate_certs: False
with_tags: True
```

Executing `ansible-inventory --list -i <filename>.vmware.yml` will create a list of VMware instances that are ready to be configured using Ansible.

RTM

Hopefully, these tips will get you thinking about how to use Ansible most effectively, but there's even more goodness in the Ansible docs. The links below can help you get the most of your Ansible experience:

- Ansible best practices
- Ansible variable precedence
- Ansible loops and control structures
- Ansible error handling
- Creating reusable Ansible Playbooks