RED HAT®
CONSULTING

redhat.

Red Hat

Ansible Engine and Ansible Tower

Preferred Practices Guide

Prepared by:

MATT NOLAN
PRINCIPAL ARCHITECT
STP - Cloud Practice
mnolan@redhat.com

Table of Contents

Purpose

This document provides Ansible Engine and Ansible Tower implementation and configuration preferred practices. It is informed by a combination of Red Hat Engineering and experience of Red Hat Consultants learned over several Ansible field engagements. Its purpose is to serve as a reference for future customer engagements.

Ansible Engine

This section details the Implementation and Configuration Preferred Practices for Ansible Engine

Sizing

- **Memory:**
  - 4GB RAM for Ansible services
  - Then 4GB RAM per 100 forks

Forks

- **Increasing the fork count:**
  - Managing Linux hosts:
    - Most tasks will run on the managed hosts and the control node will have much less of a load.
    - In this case, you can usually set forks to a much higher value, possibly closer to 100 and see performance improvements.
  - Managing network switches / routers / API:
    - In this case, playbooks may run a lot of code (modules) on the control node itself which increases the load
    - Therefore, you should raise the fork limit more judiciously

Configuring Ansible

- **Ansible Configuration File:**
  - Create an **ansible.cfg** file in the project directory from which you run Ansible commands.
  - This is the most common location used for an Ansible configuration file.
  - It is unusual to use a **~/.ansible.cfg** or **/etc/ansible/ansible.cfg** file in practice.

- **Configuring Connections to Managed Hosts:**
  - Authentication
    - Use SSH keys
    - Passwords are supported, but SSH keys with ssh-agent is recommended
  - Privilege Escalation
    - Include the **[privilege_escalation]** section in the Ansible configuration file for security and auditing reasons when connecting to remote hosts as an unprivileged user before escalating privileges to get administrative access as root.
    - Think through the security implications of whatever approach you choose for

privilege escalation. Different organizations and deployments might have different trade-offs to consider.

- To enable privilege escalation by default, set the directive **become = true** in the configuration file. Even if this is set by default, there are various ways to override it when running ad hoc commands or Ansible Playbooks. (For example, there might be times when you want to run a task or play that does not escalate privileges.)
- The become_method directive specifies how to escalate privileges. Several options are available, but the default is to use sudo.
- Likewise, the become_user directive specifies which user to escalate to, but the default is root.
- If the become_method mechanism chosen requires the user to enter a password to escalate privileges, you can set the become_ask_pass = true directive in the configuration file.
- On RHEL7, the default configuration of /etc/sudoers grants all users in the wheel group the ability to use sudo to become root after entering their password.
- One way to enable a user ('someuser' in the following example) to use sudo to become root without a password is to install a file with the appropriate directives into the /etc/sudoers.d directory (owned by root, with octal permissions 0400):

Inventories

- Site-wide defaults should be defined as a group_vars/all
  - ```
    ---
    # file: /etc/ansible/group_vars/all this is the site wide default
    ntp_server: default-time.example.com
    ```
- Regional information might be defined in a group_vars/region (overrides the higher up group based variable)
  - ```
    ---
    # file: /etc/ansible/group_vars/boston
    ntp_server: boston-time.example.com
    ```
- Specific variable for a specific host
  - ```
    ---
    # file: /etc/ansible/host_vars/xyz.boston.example.com
    ntp_server: override.example.com
    ```
- Give inventory nodes human-meaningful names rather than IPs or DNS hostnames
- Use Groups
- Use separate inventories for separate environments (Dev, QA, Prod)
- Use a single source of truth if you have it -- even if you have multiple sources Ansible can unify them
- Use meaningful names
  - 10.1.275 ----------> db1 ansible_host=10.1.2.75
  - W14301.acme.com ---------> web1 ansible_host=w14301.acme.com
    ```

- Use Dynamic Inventories
- When using multiple inventories (static / dynamic)
    - Inventory files should not depend on other inventory files or scripts in order to resolve.
    - For example, if a static inventory file specifies that a particular group should be a child of another group, it also needs to have a placeholder entry for that group, even if all members of that group come from the dynamic inventory.
    - Consider the **cloud-east**group in the following example:

      [cloud-east] *(Defined even though it's empty)*

      [servers]

      test.demo.example.com

      [servers:children]

      cloud-east

Managing Variables

- **Use a Variable Naming Standard:**
    - Avoid collisions and confusion by adding the role name to a variable as a prefix
    - Ex. - webcluster_apache_max_keepalive: 25
    - Ex - webcluster_apache_port: 80
    - Ex. - webcluster_tomcat_port: 8080

- **Defining Variables:**
    - Variables can be defined in a variety of places in an Ansible project. However, this can be simplified to three basic scope levels:
        - Global scope:
            - Variables set from the command line or Ansible configuration.
        - Play scope:
            - Variables set in the play and related structures.
        - Host scope:
            - Variables set on host groups and individual hosts by the inventory, fact gathering or registered tasks.

- **Variable Location:**
    - Find the appropriate place for your variables based on what, where and when they are set or modified
    - it is best to set a value for a variable in exactly one place to help avoid issues with variable precedence.

- **Group and Host Vars**
    - The preferred approach to defining variables for hosts and host groups is to create two

directories in the same working directory as the inventory file or directory
    - group_vars and host_vars
  - These directories contain files defining group variables and host variables, respectively.
  - The group_vars files should have matching names of the host groups to which they apply
  - The host_vars files should have matching names of the managed hosts to which they apply

- **Role Defined Variables**
  - /Default - Define variables here that are required to allow the role to work, but expected to be overridden when used
  - /Vars - Define variables here that are less likely to ever need to be overridden
  - Use a requirements.yml for shared roles

- **Misc:**
  - Separate logic (tasks) from variables and reduce repetitive patterns
  - Do not use every possibility to store variables - settle on a defined scheme and limit to as few places as possible

- **Variables and Arrays:**
  - Instead of assigning configuration data that relates to the same element (a list of packages, a list of services, a list of users, and so on), to multiple variables, use arrays.
  - One benefit of this is that an array can be browsed
  - Example:

```
users:
  bjones:
    first_name: Bob
    last_name: Jones
    home_dir: /users/bjones
 acook:
    first_name: Anne
    last_name: Cook
    home_dir: /users/acook
```

Then access like this:

```
users.bjones.first_name   # Returns 'Bob'

users.acook.home_dir     # Returns '/users/acook'
```

- Or this:

```
users['bjones']['first_name']    # Returns 'Bob'

users['acook']['home_dir']      # Returns '/users/acook'
```

- Both syntaxes are valid, but to make troubleshooting easier, Red Hat recommends that you use one syntax consistently in all files throughout any given Ansible project.

Roles

- **Scope:**
  - Keep roles focused on a specific purpose or function.
  - Instead of making one role do many things, you might write more than one role.
  - Resist creating new roles for edge configurations.
  - If an existing role accomplishes a majority of the required configuration, refactor the existing role to integrate the new configuration scenario.
  - Limit your role's dependencies on other roles. Dependencies make it harder to maintain a role, especially if it has many complex dependencies

- **Creation**
  - To create a new role, use ansible-galaxy init...then remove unneeded directories and stub files
  - Use the roles/{subdirectory} structure for roles developed for organizational clarity in a single project
  - Follow the Ansible Galaxy pattern for roles that are to be shared beyond a single project

- **Management**
  - Manage your roles in your applications repo
  - Maintain each role in its own version control repository. Ansible works well with git-based repositories.
  - As part of build process, “push” role to artifact repository
  - Use ansible-galaxy to install a role from artifact repository
  - Sensitive information, such as passwords or SSH keys, should not be stored in the role repository.
  - Sensitive values should be parameterized as variables with default values that are not sensitive.
  - Playbooks that use the role are responsible for defining sensitive variables through Ansible Vault variable files, environment variables, or other ansible-playbook options.
  - If your application requires a lot of configuration files, your roles can live with your app source code

- **Usability / Testing**
  - Use default variables to alter the role's behavior to match a related configuration scenario. This helps make the role more generic and reusable in a variety of contexts.
  - The value of any variable defined in a role's defaults directory will be overwritten if that same variable is defined:
    - in an inventory file, either as a host variable or a group variable.

        - in a YAML file under the group_vars or host_vars directories of a playbook project
        - as a variable nested in the vars keyword of a play
        - as a variable when including the role in roles keyword of a play
    - Use integration and regression testing techniques to ensure that the role provides the required new functionality and also does not cause problems for existing playbooks.
    - Reuse and refactor roles often.
    - Create and maintain README.md and meta/main.yml files to document what your role is for, who wrote it, and how to use it.

- **Tracking Changes / Updates**
    - Track the latest version of a role in your project by periodically reinstalling the role to update it.
        - This ensures that your local copy stays current with bug fixes, patches, and other features.
    - If using a third-party role in production, always specify the versionthat you want to use in order to avoid breakage due to unexpected changes.
        - If you do this, you should be periodically updating to the latest version of the role in a test environment so that you can adopt improvements and changes in a controlled manner.

Playbooks

- Use a concise description of the play's or task's purpose to name plays and tasks. The play name or task name is displayed when the playbook is executed. This also helps document what each play or task is supposed to accomplish, and possibly why it is needed.
- Include comments to add additional inline documentation about tasks.
- Make effective use of vertical white space. In general, organize task attributes vertically to make them easier to read.
- Consistent horizontal indentation is critical. Use spaces, not tabs, to avoid indentation errors.
- Set up your text editor to insert spaces when you press the Tab key to make this easier.
- Try to keep the playbook as simple as possible. Only use the features that you need.
- Keep plays and playbooks small and focused
- Several smaller and focused playbooks are better than one huge playbook full of conditionals
- Separate provision from deployment and configuration tasks

```
acme_corp/
├──── configure.yml
├──── provision.yml
└──── site.yml

$ cat site.yml
---
```

```
- include: provision.yml
- include: configure.yml
```

- Always seek out a module first
  - ansible-doc <module_name>
  - http://docs.ansible.com/ansible/modules_by_category.html
  - If you're using run commands a lot, consider writing a module
- Include the - nameoption because it helps to document your playbook, especially when it contains multiple plays.
- Use the 'state' parameter even if it's optional for a 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.

  - **Playbook Reusability:**
    - Modularize your playbook structure by using the **import_task** or **include_task** feature

```
---
- name: Install web server
  hosts: webservers
  tasks:
    - include_tasks: webserver_tasks.yml
```

    - **Common Use Cases for task files:**
    - Create various sets of tasks for creating users, installing packages, configuring services, configuring privileges, setting up access to a shared file system, hardening the servers, installing security updates, and installing a monitoring agent.
    - Each of these sets of tasks could be managed through a separate self-contained task file.
    - If servers are managed collectively by the developers, the system administrators, and the database administrators, then every organization can write its own task file which can then be reviewed and integrated by the systems manager.
    - If a server requires a particular configuration, it can be integrated as a set of tasks executed based on a conditional (that is, including the tasks only if specific criteria are met).
    - If a group of servers need to run a particular task or set of tasks, the tasks might only be run on a server if it is part of a specific host group.
    - The incorporation of plays or tasks from external files into playbooks using Ansible's import and include features greatly enhances the ability to reuse tasks and playbooks across an Ansible environment.
    - To maximize the possibility of reuse, these task and play files should be as generic as possible.

- Variables should be used to parameterize play and task elements to widen the scope of the application of tasks and plays.
- Instead of hard coding the value of a package to install:

```
---
 - name: Install the httpdpackage
   yum:
       name: httpd
       state: latest

 - name: Start the httpdservice
   service:
      name: httpd
      enabled: true
      state: started
```

- Use variables, so the task file can also be used to deploy and configure other software and their services:

```
---
 - name: Install the {{ package }}package
   yum:
       name: "{{ package }}"
       state: latest

 - name: Start the {{ service }}service
   service:
      name: "{{ service }}"
      enabled: true
      state: started
```

- Subsequently, when incorporating the task file into a playbook, you define the variables to use for the task execution as follows:

```
 tasks:
    - name: Import task file and set variables
      import_tasks: task.yml
      vars:
         package: httpd
         service: service
```

Privilege Access

- Use the 'become' method so Ansible scripts are executed via sudo (sudo is easy to track)

- Ansible can be run as root only, but login and security reasons often request non-root access
- Create an Ansible only user
- Don't try to limit sudo rights to certain commands - Ansible does not work that way!
- Use smoke tests before starting a service

```
- name: check for proper response
  uri:
    url: http://localhost/myapp
    return_content: yes
    register: result
    until: '"Hello World" in result.content'
    retries: 10
    delay: 1
```

Templates (Jinja2)

- Avoid managing variables in a template
- Avoid extensive and intricate conditionals
- Avoid conditional logic based on hostnames
- Avoid complex nested iterations
- Label template output files as being generated by Ansible
- Consider using the ansible_managed** variable with the comment filter
  {{ ansible_managed | comment }}

Troubleshooting

- Ansible provides switches for CLI interaction
  - -vvvv
  - --step
  - --check
  - --diff
  - --start-at-task

- Use the power of included options:
  - --list-tags
  - --list-hosts
  - --syntax-check
  - --check
  - --step
  - --list-tasks

- References:
  - debug – Print statements during execution – Ansible Documentation
  - Testing Strategies -- Ansible Documentation

Managing Rolling Updates

- Use the **serial** keyword to run the hosts through the play in batches
- Each batch of hosts will be run through the entire play before the next batch is started
- This controls how many servers have the service interruptions at one time if an update needs to restart a service
- This is also useful if something goes wrong and the play fails for the first batch of hosts processed
- The playbook will abort and the remaining hosts will not be run through the play
- The **serial** keyword can also be specified as a percentage. This percentage is applied to the total number of hosts in the play to determine the rolling update batch size.

Managing Secrets

- **Use Ansible Vault:**
  - Use the view subcommand to view the file's contents without making changes
  - The edit subcommand rewrites the file, so you should only use it when making changes
  - This can have implications when the file is kept under version control

- **Variable File Management:**
  - To simplify management, set up your Ansible project so that sensitive variables and all other variables are kept in separate files.
  - The files containing sensitive variables can then be protected with the ansible-vault command.
  - The preferred way to manage group variables and host variables is to create directories at the playbook level
  - The group_vars directory normally contains variable files with names matching host groups to which they apply.
  - The host_vars directory normally contains variable files with names matching host names of managed hosts to which they apply.
  - However, instead of using files in group_vars or host_vars, you also can use directories for each host group or managed host.
  - Those directories can then contain multiple variable files, all of which are used by the host group or managed host.
  - For example:
    - In the following project directory for playbook.yml, members of the webservers host group uses variables in the group_vars/webservers/vars file, and demo.example.com uses the variables in both host_vars/demo.example.com/vars and host_vars/demo.example.com/vault:

```
├── ansible.cfg
├── group_vars
│   └── webservers
│   └── vars
├── host_vars
│   └── demo.example.com
│    ├── vars
│    └── vault
├── inventory
└── playbook.yml
```

- The advantage is that most variables for demo.example.com can be placed in the vars file, but sensitive variables can be kept secret by placing them separately in the vault file.
- The administrator can use ansible-vault to encrypt the vault file, while leaving the vars file as plain text.

- Playbook variables (as opposed to inventory variables) can also be protected with Ansible Vault.
- Sensitive playbook variables can be placed in a separate file which is encrypted with Ansible Vault and which is included in the playbook through a vars_files directive.
- This can be useful, because playbook variables take precedence over inventory variables.

- **Speeding up Vault Operations:**
    - By default, Ansible uses functions from the python-crypto package to encrypt and decrypt vault files.
    - If there are many encrypted files, decrypting them at startup may cause a perceptible delay.
    - To speed this up, install the python-cryptography package: `'python-cryptography'`
    - When using multiple vault passwords with your playbook, make sure that each encrypted file is assigned a vault ID, and that you enter the matching password with that vault ID when running the playbook.
    - This ensures that the correct password is selected first when decrypting the vault-encrypted file, which is faster than forcing Ansible to try all the vault passwords you provided until it finds the right one.

Ansible Tower

This section details the Implementation and Configuration Preferred Practices for Ansible Engine
As of: **version: 3.2**

Sizing

- **Memory:**
  - 4GB RAM for Ansible services
  - Then 4GB RAM per 100 forks

- **Disk Storage**
  - At least 20GB of hard disk space is required for Ansible Tower, and 10GB of this space must be available for the /var directory

- **For OpenShift Installations**
  - If you are deploying Ansible Tower as a pod to a Red Hat OpenShift cluster, the cluster will need 6GB of memory and 3 x CPU cores per pod.
  - For more information on the container-based installation process, see OpenShift Deployment and Configuration in the Ansible Tower Administration Guide.

- **Notes:**
  - Actual RAM requirements vary based on how many managed hosts Tower is expected to manage simultaneously (which is controlled by the forks parameter in the job template or the system ansible.cfg file).
  - To avoid possible resource conflicts, Red Hat recommends 4 GB of memory per 100 forks.
  - For example:
    - If forks is set to 100, 4GB of memory is recommended; if forks is set to 400, 16 GB of memory is recommended.
    - A larger number of hosts can of course be addressed, though if the fork number is less than the total host count, more passes across the hosts are required.
    - Memory limitations can be avoided when using rolling updates (serial) option or when using the provisioning callback system built into Tower, where each system requesting configuration enters a queue and is processed as quickly as possible; or in cases where Tower is producing or deploying images such as AMIs.
    - All of these are great approaches to managing larger environments.

General

- **Content:**
  - Treat Ansible content like code and put it in version control
  - Start simple. Roles, playbooks and inventory in one place, then push out the roles you want to share into individual repos and manage with requirements.yml.

- **Style Guides:**
  - Use style guides to promote standards, reusability, and longevity of Ansible content.
  - Style can be enforced through pre-commit hooks or code reviews
    - Tagging standards
    - Whitespace standards
    - Naming of Tasks, Plays, Variables, and Roles standards
    - Directory Layouts standards

- **Inventories:**
  - Use smart inventories or smaller targeted inventories so the workflow runs more efficiently
  - An inventory can be set at the workflow level or prompted for on launch
  - Re-sync inventories during plays in the workflow
  - When launched, all job templates in the workflow that have ask_inventory_on_launch=true will use the workflow level inventory.

- **Job templates**
  - Re-use of well-defined/tested job templates in a larger orchestration context
  - Job templates that do not prompt for inventory will ignore the workflow inventory and run against their own inventory.

- **Misc:**
  - In a workflow convergence scenario, set_stats data will be merged in an undefined way, so it is recommended that you set unique keys.

Inventories

- Use dynamic as much as possible

Job Templates

- Keep jobs simple, focused - as playbooks or roles
- Use labels
- For idempotent jobs, create “check” templates as well - and let them run over night
- Combine with notifications - and get feedback when a “check” failed

Workflows

- **Surveys:**
  - Job Templates in Workflows should not have a survey enabled on them individually
  - Instead, use a comprehensive survey at the workflow level that contains all the user provided variables needed by any job templates in the workflow
  - To mitigate impacting the reusability of individual job templates (*that could use surveys outside the scope of a workflow*), copy the job templates and give them unique names to identify their exclusive use within specific larger workflows.

- **Inventory Refresh Optimizations:**
  - To reduce inventory refresh run times, use smaller and more targeted inventories for specific workflows to minimize the refresh times
  - This will expedite the run time of the workflow by avoiding much larger and more comprehensive inventory refreshes

- **Projects and Inventories:**
  - Jobs that synchronize Projects or Inventories can also be incorporated into a workflow.
  - This is recommended to ensure that Project and Inventory resources are updated prior to the use of Job Templates that depend on them

Security

- **RBAC:**
    - Organizations
        - When to use a single organization
            - Use a single organization to share roles and playbooks across departmental or functional boundaries if the enterprise is smaller and it is likely that different groups can deploy across a common set of inventories and need to share playbooks.
            - For example, an operations group of an organization may already have Ansible roles for provisioning production web, database, and application servers, which the developers group should use to provision servers to prepare their development environment. Tower makes it easier for different users and groups to use existing roles and playbooks.
        - When to use separate organizations
            - For very large deployments
            - If certain departments do not deploy to certain inventories of hosts, or run certain playbooks. It is useful to categorize large numbers of Users, Teams, Projects, and Inventories under one umbrella organization based on department.
            - By using separate organizations, a collection of Users and Teams can be configured to work with only the Tower resources that they're expected to use.
- **Credentials:**
    - Assign system or machine authentication privileges to Users and Teams using Credentials
    - This will avoid exposing passwords and having to re-key the passwords if a user leaves or changes teams later on

Variables

- When using static inventories managed in the Tower, avoid creating conflicts by setting the same variables on the inventory in Tower and in the project group and host vars files.

- **Example:**
  - If a project has host_vars or group_vars files defined, they are honored, but they can't be edited in Tower
  - If the same host or group variable is defined in both the project files and in a static inventory object managed by Tower, and have different values, which value Tower uses can be unpredictable.

- **Precedence:**
  - Inventory variables can be overridden by variables with a higher precedence.
  - Extra variables defined in a job template and playbook variables both have higher precedence than inventory variables.

Tenants, Team, Users

- Provide automation to others without exposing credentials
- Let others only see what they really need
- Use personal view instead of full Tower interface

Access / Credentials

- Set up a separate user and password/key for Tower
- That way, automation can easily be identified on target machines
- The key/password can be ridiculously complicated secure
- Store key/password in a safe for emergencies

Managing Ansible Projects (Git)

- **Infrastructure as Code:**
  - Store Ansible playbooks, roles, templates and any other content used by an Ansible Tower project in a version control system like Git.
  - Use version control to implement a life cycle for the different stages of infrastructure Ansible code such as Development, QA, and Production.
  - By managing infrastructure code with a version control tool, administrators can test infrastructure code changes in noncritical development and QA environments to minimize disruptions when deployments are implemented in production environments.
  - Configure Tower to pull projects containing playbooks from a Git repository so that specific projects pull content from specific branches of the Git repository.

- **Structuring Ansible Projects in Git:**
  - Each Project in Ansible / Tower should have its own Git repository

- **Benefits:**
  - Ability to review and restore old versions of files

- Ability to compare two versions of the same file to identify changes
- Version control systems provide a record of changes that can be used as an audit trail
- Mechanisms for multiple users to collaboratively modify files, resolve conflicting changes, and merge the changes together

Managing Facts

- **Use Fact Caching to Improve Performance:**
  - Use **fact_caching** for your job templates:
    - The plays then depend on the information in the fact cache to use facts instad of running the setup module every time
    - Using fact_caching will speed up execution of the playbooks in your project that DO NOT need to gather facts
    - This can be done by setting gather_facts: no in the playbooks to prevent the setup module from running at the beginning of every play by default

  - Reference another host's facts by using the **"magic"** variable hostvars:
    - For example, a task running on the managed host servera can access the value of the fact ansible_facts['default_ipv4']['address'] for serverb by referencing the variable hostvars['serverb']['ansible_facts']['default_ipv4']['address'].
    - However, this assumes that facts have been gathered for serverbby this play or the previous plays in the same playbook
    - One good way to do this is to set up a job template / playbook that gathers and caches facts as a scheduled job.
    - That job could run a normal playbook that gathers facts, or you could set up a minimal playbook to gather facts, such as the following example:

      ```
      - name: Refresh fact cache
        hosts: all
        gather_facts: yes
      ```

- You do not need to gather facts for all your hosts at the same time. It might make sense to gather facts for smaller sets of your hosts to spread the load.
- The important thing is to make sure you gather facts for all of your hosts before they expire or become stale.
- Manage timeouts for the fact cache at a global level.

Security

- **Vault Credentials:**
  - For projects that have vaulted files included in them, create a Vault Credential for Tower to use to decrypt those files (such as playbooks or included variable files containing secrets)

- Then add the matching vault credential to the Job Template using that Project with vaulted files in addition to any machine credentials or other credentials that Project needs.

Troubleshooting

- Use #>ansible-tower-service
  - Check status of all the core components of Tower
    - Nginx
    - Rabbitmq
    - PostgreSQL
    - Supervisord
    - Memcache

- Look at the main tower log at:
  - /var/log/tower/tower.log