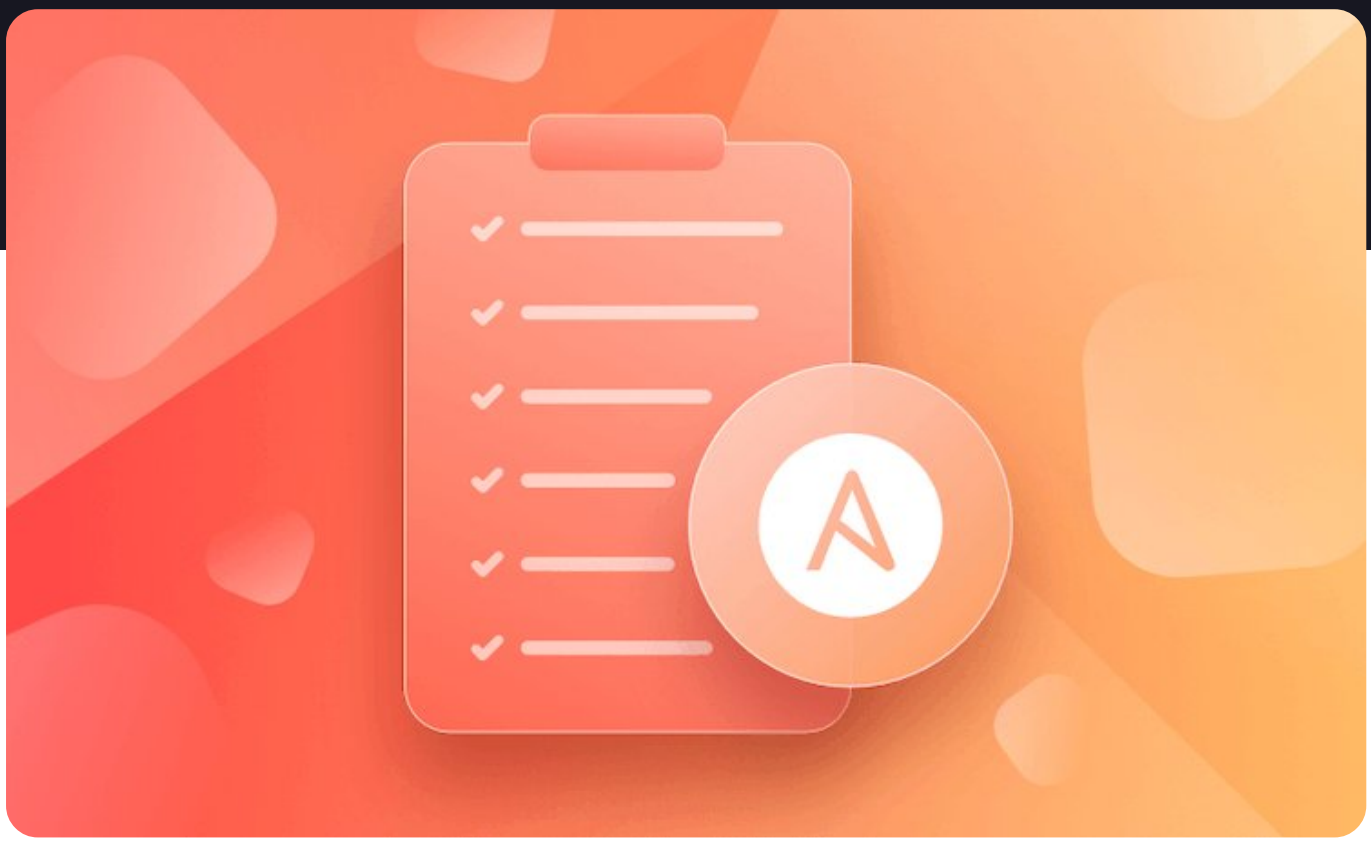
spacelift

ANSIBLE

44 Ansible Best Practices to Follow [Tips & Tricks]

Ioannis Moustakis

26 Sep 2022 · 12 min read



Ansible is one of the most used open-source software tools for configuration management, software provisioning, and application deployment of cloud or on-premises environments. This article will look into best practices for setting up Ansible projects and suggest approaches to deal with Ansible's internals effectively.

6. Execution and Deployments Best Practices and Tricks

If you are new to Ansible, take a look at this Ansible Tutorial for Beginners.

1. Generic & Project Structure Best Practices

In this part, we will examine and discuss general best practices and recommendations for organizing your projects and getting the most out of Ansible.

Generic Best Practices

Prefer YAML instead of JSON: Although Ansible allows JSON syntax, using YAML is preferred and improves the readability of files and projects. (See: YAML vs. JSON)

Use consistent whitespaces: To separate things nicely and improve readability, consider leaving a blank line between blocks, tasks, or other components.

Use a consistent tagging strategy: Tagging is a powerful concept in Ansible since it allows us to group and manage tasks more granularly. Tags provide us with the option to add fine-grained controls to the execution of tasks.

Add comments: When you think a further explanation is needed, feel free to add a comment explaining the purpose and the reason behind plays, tasks, variables, etc.

Use a consistent naming strategy: Before starting to set up your Ansible projects, consider applying a consistent naming convention for your tasks (always name them), plays, variables, roles, and modules.

Define a style guide: Following a style guide might be helpful to be consistent with the above suggestions. After all, how can we be consistent if we don't have a style guide to follow and enforce? If you are searching for inspiration, look at this style guide by OpenShift.

Keep it simple: Ansible provides many options and advanced features, but that doesn't

mean we have to use all of them. Find the Ansible parts and mechanics that fit your use case and keep your Ansible projects as simple as possible. For example, begin with a simple playbook and static inventory and add more complex structures or refactor later according to your needs.

Store your projects in a Version Control System (VCS): Keep your Ansible files in a code repository and commit any new changes regularly.

Don't store sensitive values in plain text: For secrets and sensitive values, use Ansible Vault to encrypt variables and files and protect any sensitive information.

Test your ansible projects: Use linting tools like Ansible Lint and add testing steps in your CI/CD pipelines for your Ansible repositories. For testing Ansible roles, have a look at Molecule. To test inputs or verify custom expressions, you can use the assert module.

Directory Organization

Take a look at this example of how a well-organized Ansible directory structure looks:

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

group_vars/
   group1.yml             # variables for particular groups
   group2.yml
host_vars/
  host1.yml               # variables for particular systems
  host2.yml

library/                  # Store here any custom modules (optional)
module_utils/             # Store here any custom module_utils to support modules (
filter_plugins/           # Store here any filter plugins (optional)

master.yml                # master playbook
```

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

roles/
   example_role/               # 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 jinja2
       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

   monitoring/              # same kind of structure as "common" was above, done fo
```

Inventory Best Practices

Use inventory groups: Group hosts based on common attributes they might share (geography, purpose, roles, environment).

Separate Inventory per Environment: Define a separate inventory file per environment (production, staging, testing, etc.) to isolate them from each other and avoid mistakes by targeting the wrong environments.

Dynamic Inventory: When working with cloud providers and ephemeral or fast-changing environments, maintaining static inventories might quickly become complex. Instead, set

up a mechanism to synchronize the inventory dynamically with your cloud providers.

Leverage Dynamic Grouping at runtime: We can create dynamic groups using the `group_by` module based on a specific attribute. For example, group hosts dynamically based on their operating system and run different tasks on each without defining such groups in the inventory.

```
- name: Gather facts from all hosts
  hosts: all
  tasks:
    - name: Classify hosts depending on their OS distribution
      group_by:
        key: OS_{{ ansible_facts['distribution'] }}

# Only for the Ubuntu hosts
- hosts: OS_Ubuntu
  tasks:
    - # tasks that only happen on Ubuntu go here

# Only for the CentOS hosts
- hosts: OS_CentOS
  tasks:
    - # tasks that only happen on CentOS go here
```

Learn more about Ansible inventory: Working with Ansible Inventory – Basics and Use Cases.

2. Plays & Playbooks Best Practices

In this part, we will list and discuss best practices regarding using plays & playbooks, two of the most basic components of Ansible projects.

Always mention the state of tasks: To make your tasks more understandable, explicitly set the state parameter even though it might not be necessary due to the default value.

Place every task argument in its own separate line: This point is in line with the general approach of striving for readability in our Ansible files. Check the examples below.

This works but isn't readable enough:

```
- name: Add the user {{ username }}
   ansible.builtin.user: name={{ username }} state=present uid=999999 generate_ssh_
   become: yes
```

Use this syntax instead, which improves a lot the readability and understandability of the tasks and their arguments:

```
- name: Add the user {{ username }}
   ansible.builtin.user:
     name: "{{ username }}"
     state: present
     uid: 999999
     generate_ssh_key: yes
   become: yes
```

Use top-level playbooks to orchestrate other lower-level playbooks: You can logically group tasks, plays, and roles into low-level playbooks and use other top-level playbooks to import them and set up an orchestration layer according to your needs. Have a look at this example for inspiration.

Use block syntax to group tasks: Tasks that relate to each other and share common attributes or tags can be grouped using the **block** option. Another advantage of this option is easier rollbacks for tasks under the same block.

```yaml
- name: Install, configure, and start an Nginx web server
  block:
    - name: Update and upgrade apt
      ansible.builtin.apt:
        update_cache: yes
        cache_valid_time: 3600
        upgrade: yes

    - name: "Install Nginx"
      ansible.builtin.apt:
        name: nginx
        state: present

    - name: Copy the Nginx configuration file to the host
      template:
        src: templates/nginx.conf.j2
        dest: /etc/nginx/sites-available/default

    - name: Create link to the new config to enable it
      file:
        dest: /etc/nginx/sites-enabled/default
        src: /etc/nginx/sites-available/default
        state: link

    - name: Create Nginx directory
      ansible.builtin.file:
        path: /home/ubuntu/nginx
        state: directory

    - name: Copy index.html to the Nginx directory
      copy:
        src: files/index.html
        dest: /home/ubuntu/nginx/index.html
      notify: Restart the Nginx service
  when: ansible_facts['distribution'] == 'Ubuntu'
  tags: nginx
  become: true
  become_user: root
```

Use handlers for tasks that should be triggered: Handlers allow a task to be executed

after something has changed. This handler will be triggered when there are changes to `index.html` from the above example.

```
handlers:
  - name: Restart the Nginx service
    service:
      name: nginx
      state: restarted
    become: true
    become_user: root
```

Check out the Working with Ansible Playbooks blog post for more details about playbooks.

3. Variables Best Practices

Variables allow users to parametrize different Ansible components and store values we can reuse throughout projects. Let's look at some best practices and tips on using Ansible variables.

Always provide sane defaults for your variables: Set default values for all groups under `group_vars/all`. For every role, set default role variables in `roles/<role_name>/defaults.main.yml`.

Use groups_vars and host_vars directories: To keep your inventory file clean, prefer setting group and hosts variables in the `groups_vars` and `host_vars` directories.

Add the role's name as a prefix to variables: Try to be explicit when defining variable names for your roles by adding a prefix with the role name.

```
nginx_port: 80
```

```
apache_port: 8080
```

Keep your variable's setup simple: Many options are available for setting variables in Ansible. Using all of them isn't probably the way to go unless you have specific needs. Pick the ones more appropriate for your use case and keep it as simple as possible.

Check out the How to Use Different Types of Ansible Variables blog post for more pointers and tips.

4. Modules Best Practices

This part will display some tips and best practices for using Ansible modules efficiently in tasks.

Keep local modules close to playbooks: Use each Ansible project's `./library` directory to store relevant custom modules. Playbooks that have a `./library` directory relative to their path can directly reference any modules inside it.

Avoid command and shell modules: It's considered a best practice to limit the usage of `command` and `shell` modules only when there isn't another option. Instead, prefer specialized modules that provide idempotency and proper error handling.

Specify module arguments when it makes sense: Default values can be omitted in many module arguments. To be more transparent and explicit, we can opt to specify some of these arguments, like the `state` in our playbook definitions.

Prefer multi-tasks in a module over loops: The most efficient way of defining a list of similar tasks, like installing packages, is to use multiple tasks in a single module.

```
- name: Install Docker dependencies
  ansible.builtin.apt:
     name:
```

```
      - curl
      - ca-certificates
      - gnupg
      - lsb-release
    state: latest
```

Document and test your custom modules: Every custom module should include examples, explicitly document dependencies, and describe return responses. New modules should be tested thoroughly before releasing. You can create testing roles and playbooks to test your custom modules and validate different test cases.

For more details about using modules and writing your own custom modules, check the Ansible Modules – How to Use Them Efficiently blog post.

5. Roles Best Practices

Ansible roles enable reusability and sharing of code efficiently while they provide a well-structured framework for configuring and setting our projects. This part will examine some best practices and tips for creating well-defined roles.

Follow the Ansible Galaxy Role Directory structure: Leverage the `ansible-galaxy init <role_name>` command to generate a default role directory layout according to Ansible Galaxy's standards.

Keep your roles single-purposed: Each role should have a separate responsibility and distinct functionality to conform with the separation of concerns design principle. Separate your roles based on different functionalities or technical domains.

Try to limit role dependencies: By avoiding many dependencies in your roles, you can keep them loosely coupled, develop them independently, and use them without managing complex dependencies between them.

Prefer import_role or include_role: To better control the execution order of roles and tasks, prefer using `import_role` or `include_role` over the classic `roles` option.

Do your due diligence for Ansible Galaxy Roles: When downloading and using content and roles from Galaxy, do your due diligence, validate their content, and pick roles from trustworthy contributors.

Store Galaxy roles used locally: To avoid depending on the upstream of Ansible Galaxy, you can store any roles from Galaxy to your code repositories and manage them as part of your project.

6. Execution and Deployments Best Practices and Tricks

Ansible provides many controls and options to orchestrate execution against hosts. In this part, we will explore tips and tricks on optimally controlling Ansible execution based on our needs.

Test changes in staging first: Having a staging or testing environment to test your tasks before production is a great way to validate that your changes have the expected outcome.

Limit task execution to specific hosts: If you want to run a playbook against specific hosts, you can use the `--limit` flag.

Limit tasks execution to specific tasks based on tags: In case you need to run only specific tasks from a playbook based on tags, you can define which tags to be executed with the `--tags` flag.

Validate which tasks will run before executing: You can use the `--list-tasks` flag to confirm which tasks would be run without actually running them.

Validate against which hosts the playbook will run: You can use the `--list-hosts` flag to confirm which hosts will be affected by the playbook without running it.

Validate which changes will happen without making them: Leverage the `--check` flag to predict any changes that may occur. Combine it with `--diff` flag to show differences in changed files.

Start at a specific task: Use the `--start-at-task` flag to start executing your playbook at a particular task.

Use Rolling Updates to control the number of target machines: By default, Ansible attempts to run the play against all hosts in parallel. To achieve a rolling update setup, you can leverage the serial keyword. Using this keyword, you can define how many to hosts the changes can be performed in parallel.

Control playbook execution strategy: By default, Ansible finishes the execution of each task on all hosts before moving to the next task. if you wish to select another execution

strategy, have a look at this guide.

How Spacelift Can Help You With Ansible Projects

Spacelift's vibrant ecosystem and excellent GitOps flow can greatly assist you in managing and orchestrating Ansible. By introducing Spacelift on top of Ansible, you can then easily create custom workflows based on pull requests and apply any necessary compliance checks for your organization. Another great advantage of using Spacelift is that you can manage different infrastructure tools like Ansible, Terraform, Pulumi, AWS CloudFormation, and even Kubernetes from the same place and combine their Stacks with building workflows across tools.

Spacelift is currently running a closed beta of Ansible. If you want to learn more, look here.

If you wish to be part of the beta version testing and provide feedback, sign up here!

Key Points

In this blog post, we delved into best practices, tips, and tricks for operating and configuring Ansible projects. We explored approaches for structuring our Ansible projects and roles and set different configuration options regarding the inventory and variables. Lastly, we examined various tips for controlling our playbook's execution and deployments.

Thank you for reading, and I hope you enjoyed this article as much as I did.

Written by

Ioannis Moustakis

Ioannis is a Cloud Architect with a background in DevOps & Site Reliability Engineering interested in Cloud Infrastructure, Automation, CI/CD Pipelines & Containerization. In his free time, he curates a personal blog at devopsmadness.com and moustakisioannis.medium.com. Opinions expressed are solely his own.