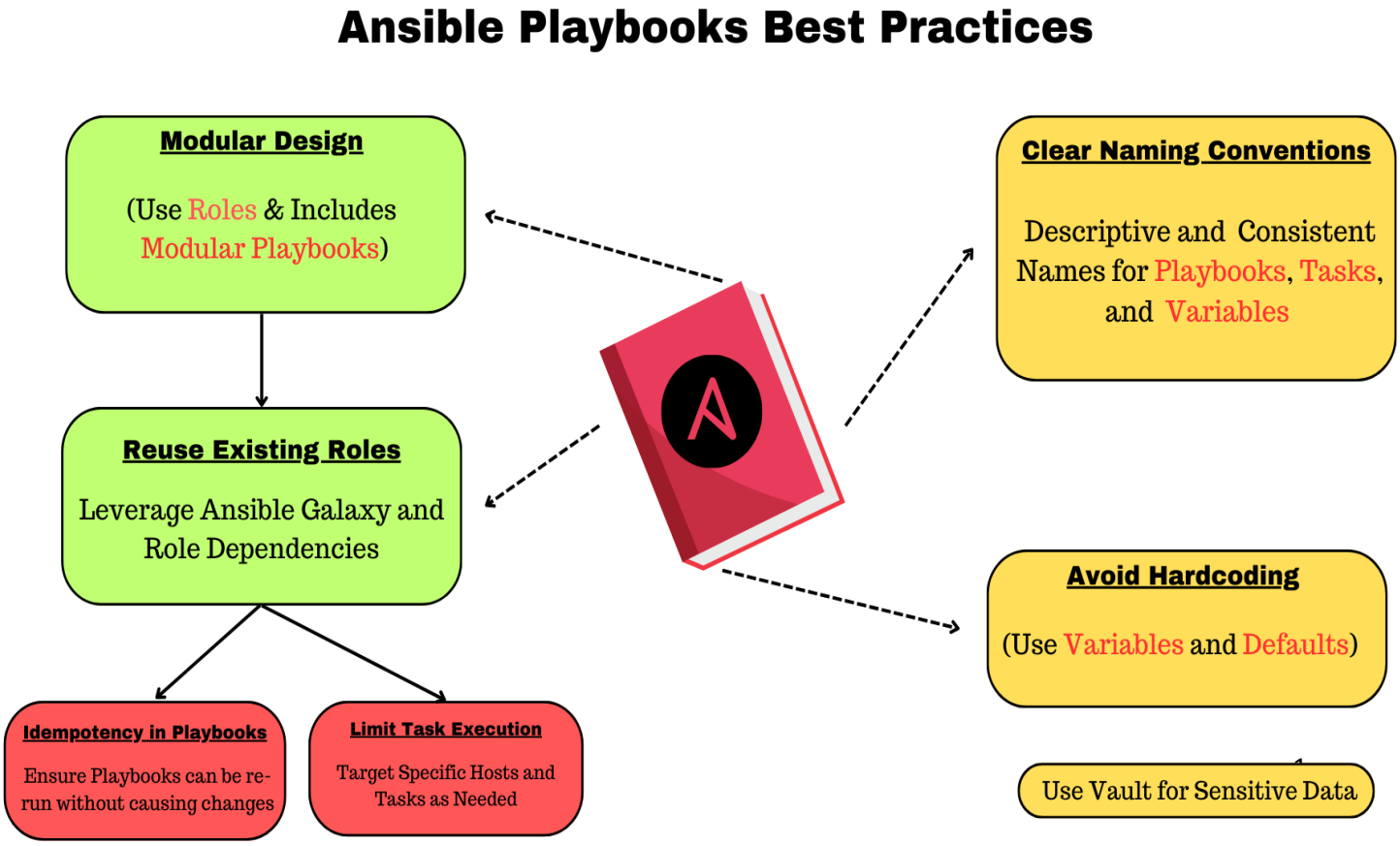
DevOps Shack

Top 12 Ansible Playbooks Best Practices

Ansible is a powerful automation tool that uses YAML-based Playbooks to define the desired state of systems and applications. Writing effective Playbooks is essential for maintainability, reusability, and ensuring predictable behavior across various environments.



1. Clear Naming Conventions

- Use meaningful, descriptive names for playbooks, tasks, variables, and files.
- Follow consistent naming patterns (e.g., lower case with underscores).

Example:

```
---
- name: Install and configure Apache
  hosts: webservers
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
```

Best Practices:

- Always name tasks so they clearly state what they do.
- Variable names should be consistent and clear.

2. Modular Design (Use Roles & Includes)

- Organize playbooks into reusable roles for better scalability and structure.
- Roles help to break large playbooks into smaller, manageable pieces.

Directory Structure:

```
├── site.yml
├── roles/
|   ├── apache/
|   |   ├── tasks/
```

```
|   |   |   └── main.yml
|   |   ├── handlers/
|   |   |   └── main.yml
|   |   ├── templates/
|   |   └── files/
```

Example:

```
---
- hosts: webservers
  roles:
    - role: apache
```

Best Practices:

- Break down tasks into logical roles (e.g., apache, mysql, security).
- Use includes and imports to split large playbooks.

3. Avoid Hardcoding (Use Variables and Defaults)

- Define variables to avoid hardcoding values and increase flexibility.
- Group variables in vars files or use group_vars for specific environments.

Example:

```
---
- name: Install a specific version of Apache
  hosts: webservers
  vars:
    apache_version: "2.4.29"
  tasks:
```

```
  - name: Install Apache
    apt:
      name: apache2={{ apache_version }}
      state: present
```

Best Practices:

- Use variable files to separate environment-specific variables (e.g., group_vars/, host_vars/).
- Provide default values using defaults/main.yml within roles.

4. Reuse Existing Roles

- Use publicly available roles from **Ansible Galaxy** to speed up development.
- Before creating new roles, check if community roles meet your requirements.

Example of Installing a Role:

```
ansible-galaxy install geerlingguy.apache
```

Best Practices:

- Always review and test Galaxy roles before integrating them.
- Customize roles as needed, but avoid modifying the role's core logic unless necessary.

5. Use Vault for Sensitive Data

- Ansible Vault should be used for securely storing sensitive information such as passwords, API keys, and other confidential data.

Example of Vault Usage:

```
ansible-vault create secrets.yml
```

Example Playbook:

yaml

Copy code

```
---
- name: Deploy application with sensitive data
  hosts: webservers
  vars_files:
    - secrets.yml
  tasks:
    - name: Use sensitive data
      debug:
        msg: "Password is {{ vault_db_password }}"
```

Best Practices:

- Use ansible-vault to encrypt sensitive variables (e.g., database passwords, credentials).
- Keep encrypted files separate and avoid including them in version control without encryption.

6. Limit Task Execution (Use Specific Hosts/Tasks)

- Avoid running unnecessary tasks on all hosts. Focus on specific hosts or use when conditions to execute tasks only when necessary.

Example:

```
---
- name: Install Apache only on webservers
  hosts: webservers
```

```
 tasks:
   - name: Install Apache
     apt:
       name: apache2
       state: present
```

Conditional Task Execution:

```
- name: Restart Apache if the configuration file has changed
  service:
    name: apache2
    state: restarted
  when: apache_config_changed
```

Best Practices:

- Use when statements to conditionally run tasks based on the environment or host variables.
- Target specific hosts using inventory grouping or hosts filters.

7. Idempotency in Playbooks

- Ensure that your playbooks can be safely re-run multiple times without causing unintended changes. This is key to the idempotency principle in Ansible.

Example:

```
- name: Ensure Apache is installed
  apt:
    name: apache2
    state: present
```

Best Practices:

- Use state-specific tasks like present or absent to maintain idempotency.
- Avoid tasks that change the system without checks (e.g., using shell or command without creates or removes).

8. Handlers and Notifications

- Use handlers to trigger actions like restarting services only when necessary, minimizing unnecessary actions.

Example:

```
---
- name: Install Apache and update configuration
  hosts: webservers
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
    - name: Update Apache configuration
      template:
        src: apache.j2
        dest: /etc/apache2/apache2.conf
      notify:
        - Restart Apache

handlers:
```

```
 - name: Restart Apache
   service:
     name: apache2
     state: restarted
```

Best Practices:

- Handlers should be used for tasks like restarting services, which should only be triggered when a task changes something (e.g., modifying a configuration file).

9. Use Tags for Selective Task Execution

- Use tags to control which tasks get executed during a playbook run. This is useful when testing or when you only want to run a subset of tasks.

Example:

```
---
- name: Setup web server
  hosts: webservers
  tasks:
    - name: Install Apache
      apt:
        name: apache2
        state: present
      tags:
        - install

    - name: Configure Apache
```

```
    template:
      src: apache.j2
      dest: /etc/apache2/apache2.conf
    tags:
      - config
```

Run with Tags:

```
ansible-playbook webserver.yml --tags "install"
```

Best Practices:

- Use tags for long playbooks to allow selective execution of tasks.
- Group tasks logically by function or component using tags.

10. Debugging and Error Handling

- Leverage Ansible's built-in tools for debugging and error handling to track down issues more efficiently.

Debugging Example:

```
- name: Print debug message
  debug:
    msg: "The value of foo is {{ foo }}"
```

Error Handling Example (Using ignore_errors):

```
- name: Ignore failure of this task
  shell: "exit 1"
  ignore_errors: yes
```

Best Practices:

- Use debug module to output variable values during development or troubleshooting.

- Be cautious with ignore_errors as it can mask issues in playbooks. Use it sparingly and only when truly needed.

11. Testing with Molecule

- Use Molecule to test your Ansible roles and playbooks in a consistent and repeatable environment.

Molecule Setup:

```
pip install molecule
molecule init role my_role
molecule test
```

Best Practices:

- Always test roles and playbooks in isolated environments before production.
- Automate testing using CI/CD pipelines to ensure code quality.

12. Documentation in Playbooks

- Well-documented playbooks are essential for maintainability, especially when multiple users are involved in managing infrastructure.

Best Practices:

- Comment complex tasks and include explanations for non-obvious logic.
- Document variables, default values, and conditions.
- Keep playbook README files up to date, explaining the purpose and usage of each playbook.

Example:

```
---
```

```
- name: Install and configure Nginx
  hosts: webservers
  tasks:
    - name: Install Nginx
      apt:
        name: nginx
        state: present
    # Install the latest version of Nginx on the server
```

Conclusion

By following these best practices, you can write Ansible Playbooks that are efficient, scalable, and easy to maintain. Use modular design, handle sensitive data securely, make use of roles, and ensure that your playbooks