Antony Kervazo-Canut

Ansible for Teenagers

SUITABLE FOR ADULTS

Introduction

Ansible is an open-source automation tool for configuration and management of IT infrastructures. It allows for the easy management of numerous servers using YAML language to describe automated tasks. Ansible operates agentlessly, meaning there is no need to install additional software on the nodes it manages.

Ansible is popular for several reasons:

- Simplicity and ease of use: The playbooks (configuration files) are written in YAML, a simple and human-readable language.
- Powerful and flexible: Ansible can handle complex tasks and adapt to various environments.
- Agentless: No need to install software on the managed nodes, simplifying management and security.
- Idempotent: Playbooks can be run multiple times without causing undesirable or unintended effects.
- Large community and support: Ansible benefits from a vast community and an abundance of resources and modules.

Ansible Installation

```
# Installation of Ansible on a Debian-based distribution
(such as Ubuntu)
sudo apt update && sudo apt install ansible

# Installation of Ansible on a Red Hat-based distribution
sudo yum install ansible

# Installation of Ansible on macOS using Homebrew
brew install ansible

# No Windows, use WSL2
```

Inventory

```
[servers]
server1 ansible_host=192.168.64.3
```

The inventory file is a configuration document used by Ansible to list and organize the hosts and server groups on which tasks and playbooks will be executed.

Ansible communicates with remote nodes using SSH for Unix/Linux systems and WinRM for Windows systems, enabling secure and efficient management of configurations and automations remotely.

Ansible Ping

```
# Testing the connection to hosts in the inventory
# ansible all -i inventory.ini -m ping
# 'ansible' is the base command, 'all' designates all hosts,
'-i' specifies the inventory file, '-m ping' uses the ping
module to test the connection
ansible all -i inventory.ini -m ping
```

This command sends a ping to all the hosts defined in inventory.ini to verify that Ansible can connect to them correctly.

```
serveur1 | SUCCESS ⇒ {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/bin/python3"
    },
    "changed": false,
    "ping": "pong"
}
```

Playbooks

An Ansible playbook is a YAML file describing the tasks to be executed on the servers.

```yaml
---
- hosts: servers
  become: yes
  tasks:
    - name: Ensure Apache is installed
      apt:
        name: apache2
        state: present
```

This playbook targets the servers group in your inventory file and executes a task to install Apache on the machines in this group.

```bash
# Exécution d'un playbook Ansible
# 'ansible-playbook' est la commande pour exécuter des playbooks, '-i' spécifie le fichier d'inventaire, 'mon_playbook.yml' est le playbook à exécuter
ansible-playbook -i inventory.ini mon_playbook.yml
```

Tasks

Tasks are the basic units of action in an Ansible playbook. They define what you want to achieve on the targeted hosts.

```yaml
---
- hosts: servers
  become: yes
  tasks:
    - name: Update all packages
      apt:
        update_cache: yes
        upgrade: dist
```

This task updates the packages on an Ubuntu server using the apt module. Ansible has a wide range of modules for various tasks. For example, to manage files, use the file module.

```yaml
---
- hosts: servers
  become: yes
  tasks:
  - name: Create a directory
    file:
      path: /my/directory
      state: directory
```

Variables

```
---
http_port: 80
max_clients: 200
```

```
---
- hosts: servers
  vars_files:
    - vars.yaml
  tasks:
    - name: Display the value of the variable 'http_port'
      debug:
        msg: "The HTTP port is {{ http_port }}"
```

```
# Running a playbook with variables passed on the command
line
# '-e' allows passing extra variables (extra-vars), here
we're redefining 'http_port'
ansible-playbook my_playbook.yml -i inventory.ini -e
"http_port=8080"
```

Facts

Facts in Ansible are automatically generated variables containing information about remote systems. They are collected by Ansible each time it connects to a target host, providing details such as the operating system, IP addresses, available disks, etc.

```
# Gathering and displaying facts for a specific host
ansible server1 -i inventory.ini -m setup
```

Facts can be used in your playbooks to condition the execution of tasks based on the characteristics of the host.

```
---
- hosts: servers
  become: yes
  tasks:
    - name: Ensure Apache is installed
      apt:
        name: apache2
        state: present
      when: ansible_os_family == "Debian"
```

Roles

Roles in Ansible organize tasks, files, templates, and variables into logical units, thereby facilitating reuse and management.

```
role_name/
├── defaults/          # Default variables for the role
│   └── main.yml       # Default variables file
├── vars/              # Other variables for the role
│   └── main.yml       # Additional variables file
├── tasks/             # Main task files
│   └── main.yml       # Main tasks file
├── handlers/          # Handlers for tasks
│   └── main.yml       # Main handlers file
├── templates/         # Jinja2 templates for configuration
│   └── template.j2    # Example template file
├── files/             # Static files required by the role
│   └── example.txt    # Example static file
└── meta/              # Role metadata
    └── main.yml       # Metadata file, including dependencies
```

```
# Creating a new role with ansible-galaxy
ansible-galaxy init role_name
```

Ansible Vault

Ansible Vault is a tool integrated into Ansible that allows for the encryption of sensitive data files for security. It is particularly useful for managing sensitive information such as passwords or secret keys in your playbooks, roles, or variable files.

```
# Creating a secret - You will be prompted to enter the vault password
ansible-vault create secret.yml

# Editing a secret
ansible-vault edit secret.yml

# Viewing a secret
ansible-vault view secret.yml

# Decrypting a secret
ansible-vault decrypt secret.yml

# Encrypting a secret
ansible-vault encrypt secret.yml

# Using a playbook containing a secret (Will ask for the password)
ansible-playbook site.yml --ask-vault-pass

# Using a playbook by passing the password as a file
ansible-playbook playbook.yml --vault-password-file /path/to/password_file
```

Ansible Vault Usage

To use encrypted variables in your playbooks, you must first include the encrypted secrets file. Use the vars_files directive in your playbook to specify the encrypted file.

```
db_password: supersecretpassword123
api_key: "123456789abcdef"
```

```
---
- hosts: all
  vars_files:
    - secrets.yml

  tasks:
    - name: Display the API key
      debug:
        msg: "The API key is {{ api_key }}"
```

Ansible Galaxy

Ansible Galaxy is a platform where the community shares reusable roles : https://galaxy.ansible.com/ui/

```
# Install a role for Nginx from Ansible Galaxy
ansible-galaxy install geerlingguy.nginx

# Install a role for MySQL from Ansible Galaxy
ansible-galaxy install geerlingguy.mysql
```

This playbook targets two groups of hosts: web servers for the Nginx role and db servers for the MySQL role.

```
---
- hosts: web_servers
  become: yes
  roles:
    - geerlingguy.nginx

- hosts: db_servers
  become: yes
  roles:
    - geerlingguy.mysql
```

Errors

Sometimes, you may want to continue the execution of the playbook even if a task fails. You can use ignore_errors.

```yaml
---
- hosts: servers
  become: yes
  tasks:
    - name: Task that might fail
      command: /a/command/that/might/fail
      ignore_errors: yes
```

You can control the conditions under which a task is considered to have failed or succeeded by using failed_when and changed_when. Here, if the word "ERROR" is present, the task fails.

```yaml
---
- hosts: servers
  become: yes
  tasks:
    - name: Execution of a script that always returns 0
      command: /my/script.sh
      register: script_result
      failed_when: "'ERROR' in script_result.stdout"
      changed_when: script_result.rc != 0
```

Error Management

Blocks allow grouping multiple tasks together and managing errors collectively.

In this example, if a task in the block block fails, the rescue block is executed. The always block executes in all cases, whether the block block succeeds or fails.

```
---
- hosts: servers
  become: yes
  tasks:
    - name: Task block
      block:
        - debug:
            msg: 'I am about to run a task'
        - command: /a/command/that/might/fail
      rescue:
        - debug:
            msg: 'An error occurred while executing the
command'
      always:
        - debug:
            msg: 'This message always displays after the
block, success or failure'
```

Retries / Until

Sometimes, you may want to retry a task that failed after a short delay.

```yaml
---
- hosts: servers
  become: yes
  tasks:
    - name: Retry a task until it succeeds
      command: /a/temporarily/unstable/command
      register: result
      until: result.rc == 0
      retries: 5
      delay: 10
```

Here, the task is repeated up to 5 times with a 10-second delay between each attempt, until it succeeds.

Assertions

Use assertions to perform checks before executing critical tasks.

```
---
- hosts: servers
  become: yes
  tasks:
    - name: Verify that the variable 'my_parameter' is
defined
      assert:
        that:
          - my_parameter is defined
        fail_msg: "'my_parameter' is not defined"
        success_msg: "'my_parameter' is defined"
```

This task ensures that the variable my_parameter is defined before proceeding.

Handlers

Handlers are special tasks that run in response to a change. For example, restarting a service after it has been updated.

```yaml
---
- hosts: servers
  become: yes
  tasks:
    - name: Install a package
      apt:
        name: my_package
        state: latest
      notify:
        - restart my_service
  handlers:
    - name: restart my_service
      service:
        name: my_service
        state: restarted
```

Debug

Use the debug module to display variable values or custom messages.

```yaml
---
- hosts: servers
  become: yes
  tasks:
    - name: Display the value of a variable
      debug:
        var: my_variable
```

Run playbooks in verbose mode (-v, -vv, or -vvv for more details) to get additional information.

```bash
# Running a playbook in verbose mode
ansible-playbook my_playbook.yml -i inventory.ini -v
```

Molecule

Molecule provides a framework for systematically testing Ansible roles, using containers or virtual machines to create a test environment. It allows you to verify that your roles work as expected across different platforms and configurations.

```
# Installing Molecule with Docker support
pip install 'molecule[docker]'

# Initializing a new role with Molecule
molecule init role -r role_name
```

Molecule allows writing tests to verify that Ansible roles work as intended. These tests are usually written using the testinfra module or ansible itself for the verification phase.

```
molecule/
│   └── default/
│       ├── converge.yml
│       ├── molecule.yml
│       └── verify.yml
```

Molecule with Testinfra

Testinfra is a Python library used for testing your servers' infrastructure. It allows writing unit tests to verify if services, packages, files, etc., are in the expected state on your servers. Testinfra can be used with Molecule to test Ansible roles, ensuring the role applies the desired configuration on a server or container.

```yaml
---
dependency:
  name: galaxy
driver:
  name: docker
platforms:
  - name: instance-ubuntu
    image: "docker.io/library/ubuntu:20.04"
    pre_build_image: true
provisioner:
  name: ansible
  playbooks:
    converge: converge.yml
verifier:
  name: testinfra
  options:
    v: 2
  directory: ../tests/
```

molecule.yml

Molecule with Testinfra

With Molecule using Testinfra as the verifier, tests are not defined in a verify.yml file but rather in separate Python test files, such as test_default.py.

```python
def test_nginx_installed(host):
    nginx = host.package("nginx")
    assert nginx.is_installed

def test_nginx_running_and_enabled(host):
    nginx = host.service("nginx")
    assert nginx.is_running
    assert nginx.is_enabled
```

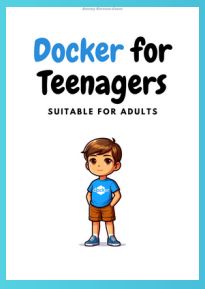
tests/test_default.py

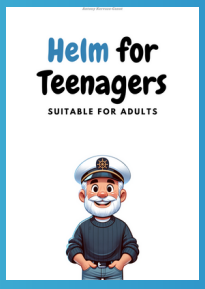
Molecule will create a Docker instance for testing, apply the role, run the tests, and then destroy the Docker instance.

```
molecule test
```

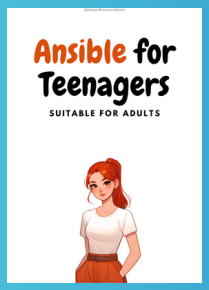
In the same collection

Container Orchestration and Management





Infrastructure as Code



Security & Secrets Management

ANTONYCANUT

ANTONY KERVAZO-CANUT