Ansible Playbooks

Ansible Fundamentals

Agenda

- YAML overview
- Modules, Tasks, Plays, Playbooks
- General Playbook Structure
- Task Results
- Validating the Result
- Writing Idempotent Tasks

YAML overview

Ansible Playbooks

YAML Overview: Basics

- Format
  - YAML stands for "YAML Ain't Markup Language."
  - It's a human-readable data serialization format
- Indentation
  - Uses spaces (not tabs) for indentation, which denotes hierarchy
  - Most of the issues with YAML is about identation
- Case Sensitive
  - YAML is case sensitive

YAML Overview: Document Start/End

- Start
  - An optional `---` at the beginning indicates the start of a YAML document
- End:
  - An optional `...` at the end indicates the end of a YAML document
- If you want to add more than one playbook on a file, you need to use the start element (`---`) to separate the objects

YAML Overview: Data Types

- Scalars
  - Single values, which can be strings, numbers, or booleans
- Mappings
  - Key-value pairs, similar to dictionaries or hashes in other languages
  - Denoted with **`key: value`** format
- Lists
  - Ordered sequences of values
  - Each item in a list is denoted with a - (dash) followed by a space

YAML Overview: Strings and comments

- Quotation
  - Strings can be written with or without quotes
  - However, for strings containing special characters or reserved words, it's safer to use single or double quotes
- Multiline
  - Use the **>** character for folded style (newlines become spaces)
  - Use **|** for literal style (newlines are preserved)
- Comments
  - Use **#** to add comments
  - Everything after **#** on that line is a comment.

- More here: https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html#yaml-syntax

Modules, Tasks, Plays, Playbooks

Ansible Playbooks

Modules

- Modules are the units of work in Ansible
- They are like command-line tools but can be run directly or through a playbook
- Each module is designed to accomplish a specific task, such as managing packages, creating users, or interacting with APIs
- Relationship
  - Modules are the building blocks that tasks use to perform actions.

Tasks

- Tasks define a single action that will be executed on the target host
- Each task calls an Ansible module with specific arguments
- Relationship
  - A task is essentially an instance of a module with specific parameters
  - Multiple tasks together form the actions in a play.

Play

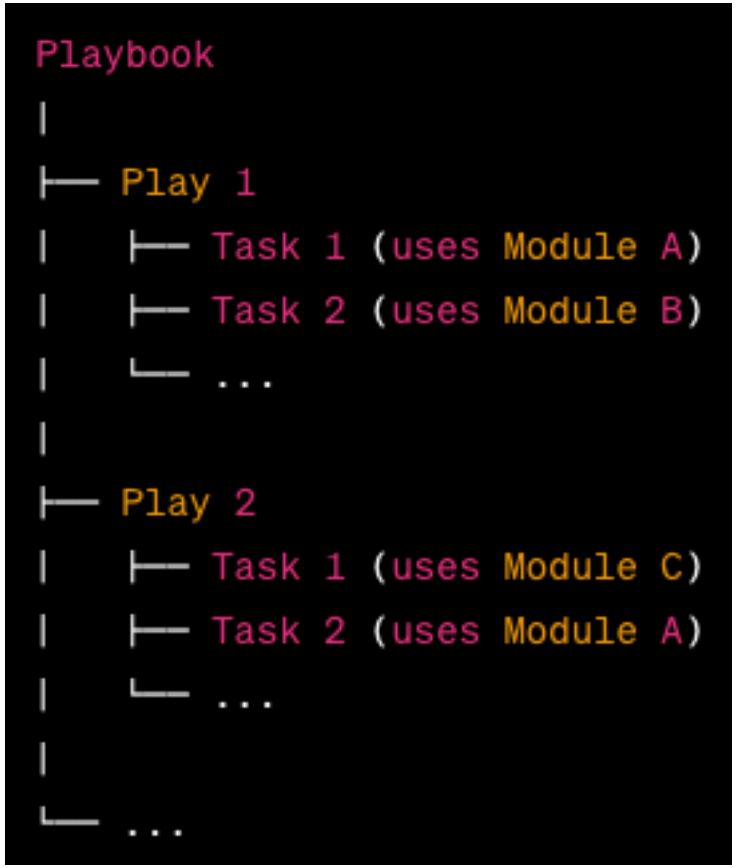
- A play is a set of tasks that will be run on a particular set of hosts in a specific order
- It defines which hosts from the inventory the tasks should run on and sets variables that can be used in the tasks.
- Relationship
  - Plays organize tasks
  - A single playbook can contain multiple plays, allowing for different sets of tasks to be run on different hosts or groups of hosts.

Playbook

- A playbook is a YAML file that contains one or more plays
- It provides a script-like experience, where multiple plays are executed in order, each with its set of tasks.
- Relationship
  - The playbook is the top-level component
    - It orchestrates the execution of plays, which in turn run tasks that call upon modules

Modules, Tasks, Plays, Playbooks

- A **module** is a tool that performs a specific action.
- A **task** uses a module to execute that action with specific parameters.
- A **play** is a collection of tasks executed on a group of hosts.
- A **playbook** orchestrates multiple plays, defining the broader automation workflow.



Single-Play Playbook

- Simplicity
  - Easier to read and understand, especially for newcomers or for quick tasks.
- Specificity
  - Ideal for focused tasks or roles, such as deploying a single application or configuring a specific service.
- Modularity
  - Can be easily included or imported into other playbooks or roles, promoting reusability.
- Clear Execution
  - With only one play, there's a clear start and end, reducing potential confusion.

Multi-Play Playbook

- Organization
  - Allows for structuring complex workflows in a single file, with each play handling a specific part of the process.
- Sequential Operations
  - Useful when operations on one group of hosts depend on operations on another group
  - For example, setting up a database server before deploying an application that uses that database.
- Conditional Execution
  - Different plays can be conditionally executed based on the results of previous plays or external factors.
- Parallelism
  - Ansible can run tasks on multiple hosts in parallel.
  - By grouping hosts in different plays, you can achieve efficient parallel execution while maintaining a specific order where needed.

Sequential Execution

- Plays
  - In a playbook, plays are executed sequentially
  - If you have multiple plays in a playbook, the first play will run to completion on all targeted hosts before the second play starts, and so on
- Tasks
  - Within a play, tasks are also executed sequentially
  - The first task will run on all targeted hosts before the second task starts, and so on

Parallel Execution

- Hosts
  - While tasks are executed sequentially in the order they are defined, the tasks themselves run in parallel across all targeted hosts.
  - For example, if you have a task to install a package and you target 10 hosts, Ansible will (by default) try to install the package on all 10 hosts at the same time.
- Forks
  - The degree of parallelism is controlled by the **`forks`** configuration.
  - The default is usually 5, meaning Ansible will run operations on **5 hosts simultaneously**
  - Once one host completes, Ansible will start the operation on the next host.
  - You can adjust this number in the **`ansible.cfg`** file or by using the **`-f`** or **`--forks`** command-line parameter
  - Setting a higher forks value will increase parallelism, but also requires more resources on the control machine

Controlling Execution

- Serial
  - If you want to control the number of hosts executing a task simultaneously, you can use the serial keyword in your play
  - For example, **`serial: 2`** would mean that the play is executed on 2 hosts at a time. Only after both have completion you start another 2
  - This is useful for rolling updates or when you don't want to impact all hosts at once.

```
---
- name: Rolling Update Play
  hosts: webservers
  serial: 2
  tasks:
    - name: Take out of load balancer pool
      ...
    - name: Update application
      ...
    - name: Add back to load balancer pool
      ...
```

General Playbook Structure

Ansible Playbooks

Plays Structure

- **`hosts`**
  - Specifies which hosts the tasks will run on
  - Can target individual hosts, groups, or patterns
- **`vars`**
  - Defines play-level variables
  - For their usage you may reference using **`{{ var-name }}`**
- **`tasks`**
  - A list of tasks to execute in order
  - Each task calls an Ansible module.
- **`handlers`**
  - Special tasks that run at the end of a play if notified by another task.
- **`become`**
  - Allows privilege escalation, e.g., executing tasks as sudo.
- **`serial`**
  - Controls how many hosts are managed at a single time (for rolling updates).

Variables

- You can define them directly on the playbook, using group variables or host variables
- Ansible already have some built-in variables that can grant you some context variables
  - **`inventory_hostname`**
  - **`hostvars`**
  - **`ansible_play_name`**
- Complete list: https://docs.ansible.com/ansible/latest/reference_appendices/special_variables.html

Tasks Structure

- **`name:`**
  - A human-readable description of the task
- Module
  - The action to be taken, using an Ansible module
  - This parameter uses the module name directly
- **`args/vars`**
  - Arguments or parameters for the module
- **`when`**
  - Conditional statement to determine if the task should run
- **`notify`**
  - Triggers a handler if the task makes a change

Basic Playbook

```yaml
---
- name: Install and start Apache
  hosts: webservers
  tasks:
    - name: Ensure Apache is installed
      apt:
        name: apache2
        state: present
    - name: Ensure Apache is running
      service:
        name: apache2
        state: started
```

Playbook with Variables and Handlers

```yaml
- name: Deploy web application
  hosts: webservers
  vars:
    app_version: 1.2.3
  tasks:
    - name: Copy application files
      copy:
        src: /opt/myapp-{{ app_version }}/
        dest: /var/www/html/
      notify: Restart Apache
  handlers:
    - name: Restart Apache
      service:
        name: apache2
        state: restarted
```

Playbook with Conditional Tasks

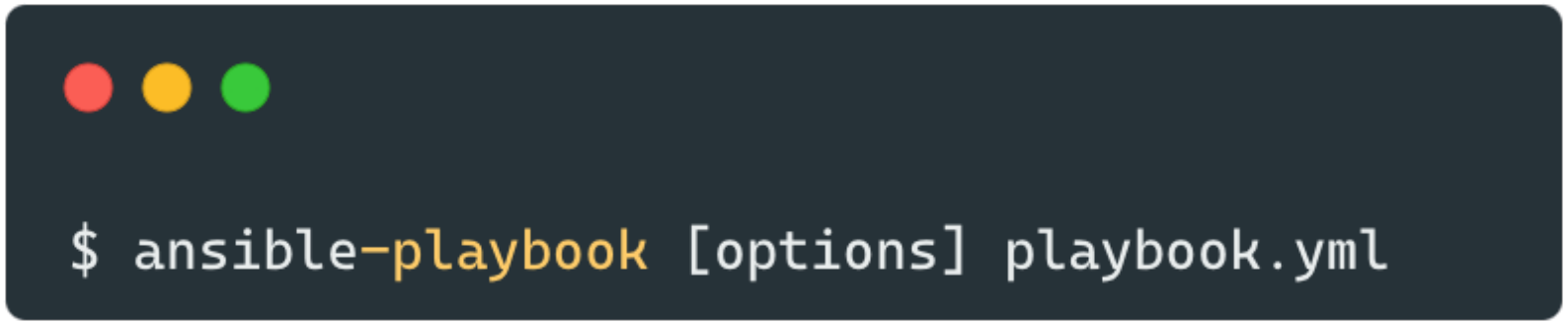
```yaml
- name: Setup environment
  hosts: all
  tasks:
    - name: Install Apache on Debian
      apt:
        name: apache2
        state: present
      when: ansible_os_family == "Debian"
    - name: Install Apache on RedHat
      yum:
        name: httpd
        state: present
      when: ansible_os_family == "RedHat"
```

Playbook with Multi-Play

```yaml
---
- name: Setup Web Servers
  hosts: webservers
  become: yes
  tasks:
    - name: Ensure Apache is installed
      apt:
        name: apache2
        state: present

    - name: Ensure Apache is running
      service:
        name: apache2
        state: started

- name: Setup Database Servers
  hosts: dbservers
  become: yes
  tasks:
    - name: Ensure MySQL is installed
      apt:
        name: mysql-server
        state: present

    - name: Ensure MySQL is running
      service:
        name: mysql
        state: started

- name: Deploy Application on Web Servers
  hosts: webservers
  tasks:
    - name: Copy application files
      copy:
        src: /opt/myapp/
        dest: /var/www/html/

- name: Create Database Schema on Database Servers
  hosts: dbservers
  tasks:
    - name: Run schema script
      command: mysql < /opt/myschema.sql
```

Execute Playbook

- Using ansible-playbook command



Common Options

- Same as ad-hoc commands
- **`-i`** or **`--inventory`**: Specify the location of the inventory file
- **`-u`** or **`--user`**: Define the remote user to execute tasks as. By default, it uses the current user
- **`-k`**: Ask for SSH password instead of using key-based authentication
- **`-b`** or **`--become`**: Allows privilege escalation (e.g., using sudo). Useful if tasks need root privileges
- **`--ask-become-pass`** or **`-K`**: Ask for privilege escalation password (e.g., sudo password)
- **`-v`** to **`-vvvv`**: Increase verbosity. More "v"s give more detailed output
- **`--check`**: Run in check mode. Ansible will not make any changes on the hosts, but will simulate the execution to show what would have changed
- **`--diff`**: Show differences when changing files. Useful with --check to see what would change

Commonly used Modules

Ansible Playbooks

User Module

- Manage user accounts: https://docs.ansible.com/ansible/latest/collections/ansible/builtin/user_module.html

```
- name: Create a user
  hosts: all
  tasks:
    - user:
        name: john
        state: present
```

Group Module

- Manage group accounts: https://docs.ansible.com/ansible/latest/collections/ansible/builtin/group_module.html

```
- name: Create a group
  hosts: all
  tasks:
    - group:
        name: developers
        state: present
```

Apt Module (Package management)

- Manage apt-packages: https://docs.ansible.com/ansible/latest/collections/ansible/builtin/apt_module.html
- You can find modules for several package managers

```
- name: Install nginx
  hosts: all
  tasks:
    - apt:
        name: nginx
        state: present
```

Service Module

- Manage services: https://docs.ansible.com/ansible/latest/collections/ansible/builtin/service_module.html

```
- name: Ensure nginx is running
  hosts: all
  tasks:
    - service:
        name: nginx
        state: started
```

Copy Module

- Copies files from the local to a location on the remote machine https://docs.ansible.com/ansible/latest/collections/ansible/builtin/copy_module.html#ansible-collections-ansible-builtin-copy-module

```
- name: Copy a file
  hosts: all
  tasks:
    - copy:
        src: /localpath/myfile.txt
        dest: /remotepath/myfile.txt
```

File Module

- Manages file properties https://docs.ansible.com/ansible/latest/collections/ansible/builtin/file_module.html#ansible-collections-ansible-builtin-file-module

```
- name: Create a directory
  hosts: all
  tasks:
    - file:
        path: /path/to/directory
        state: directory
```

Command Module

- Executes a command on a remote node https://docs.ansible.com/ansible/latest/collections/ansible/builtin/command_module.html#ansible-collections-ansible-builtin-command-module

```
- name: Run a command
  hosts: all
  tasks:
    - command:
        cmd: ls -la /path/to/directory
```

lineinfile Module

- Executes a command on a remote node https://docs.ansible.com/ansible/latest/collections/ansible/builtin/lineinfile_module.html

```
- name: Ensure a line is present in a file
  hosts: all
  tasks:
    - lineinfile:
        path: /path/to/file.txt
        line: 'This is the line we want to ensure exists.'
```

Using Playbooks

Demo

Task Results

Ansible Playbooks

Task Results

- Every time you execute a task, you get a result
- Possible results
  - OK
  - Changed
  - Failed
  - Skipped
  - Unreachable

Task Results: OK

- The task executed successfully
- The module ran without any errors, and the desired state expressed in the task is already in place on the target system
- In other words, the system was already in the desired state, so no changes were made.
- Example: If you have a task to ensure a package is installed, and the package is already installed, the task result will be "OK".

Task Results: Changed

- The task executed successfully and made changes to the target system
- The module ran without any errors, and the system was not initially in the desired state, so Ansible made the necessary changes to bring the system to that state.
- Example: If you have a task to ensure a package is installed, and the package was not initially installed, Ansible will install it, and the task result will be "changed".

Task Results: Failed

- The task did not execute successfully and encountered an error.
- An error occurred that prevented the module from completing its operation.
- This could be due to various reasons like incorrect parameters, issues on the target system, unreachable hosts, etc.
- Example: If you have a task to ensure a package is installed, but there's an issue with the package repository or network connectivity, the task might fail to install the package, resulting in a "failed" state.

Task Results: Skipped

- The task was intentionally not executed on a particular host
- Tasks can be conditionally executed based on the evaluation of a when clause
- If the condition in the when clause evaluates to false, the task will be skipped for that host
- Example: If you run the following task on a RedHat system, the result will be "Skipped"

```
tasks:
  - name: Install nginx on Debian systems
    apt:
      name: nginx
      state: present
    when: ansible_os_family == "Debian"
```

Task Results: Unreachable

- Ansible was unable to establish a connection to the target host
- This state typically indicates a fundamental communication issue between the Ansible control node and the target host
- Common reasons include network connectivity problems, incorrect SSH configurations, SSH key mismatches, host firewalls blocking access, or the target host being down
- When a host is in an "Unreachable" state, Ansible will not attempt any further tasks on that host for the duration of the playbook run

```
192.168.1.10 | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: ssh: connect to host 192.168.1.10 port 22: No route to host",
    "unreachable": true
}
```

Validating the Result

Ansible Playbooks

Validating Results

- You may validate the results of a task and use that results on following tasks
- Usually, you start to save task output to a variable
- Then you may use variable content on other tasks to print values or decide about task execution

Getting task output

- Use the **`register`** keyword to save the output of a task to a variable

```
- name: Execute a command
  command: "echo 'Hello, World!'"
  register: command_output
```

- Then you can use variable attributes as a common variable
- Each task (module) will add specific attributes
- Common attributes include
  - **`command_output.stdout`**: The standard output of the command
  - **`command_output.stderr`**: The standard error of the command
  - **`command_output.rc`**: The return code of the command
  - **`command_output.changed`**: Boolean indicating if the task made changes

Using in conditionals

- Use the **`when`** keyword to conditionally execute tasks based on the result of a previous task

```yaml
- name: Check if a file exists
  stat:
    path: /tmp/example.txt
  register: file_stat

- name: Notify if file exists
  debug:
    msg: "The file exists!"
  when: file_stat.stat.exists
```

Handling Failures Manually

- Customize when Ansible should consider a task as failed using the **`failed_when`** keyword.

```
- name: Execute a command that might fail
  command: "some-command"
  register: command_result
  failed_when: "'ERROR' in command_result.stderr"
```

Debugging outputs

- Print messages, variables, or task results for debugging purposes

```
- name: Print command output
  debug:
    msg: "The command output is {{ command_output.stdout }}"
```

Writing Idempotent Tasks

Ansible Playbooks

Understanding Idempotency

- Writing idempotent tasks is a fundamental principle in Ansible
- Ensures that running your playbook multiple times doesn't change the system state after the first run, unless the system state has changed in the meantime
- A task is idempotent if it can be applied multiple times without changing the result beyond the initial application
- Ensures consistency, avoids unintended side-effects, and makes playbooks safe to run repeatedly

Imperative Configuration

- In an imperative approach, you specify how to achieve a particular state, detailing each step
- Concentrates on the process and sequence of operations to achieve the desired result
- Offers more control and can be more flexible in certain scenarios, as you dictate the exact sequence of operations.
- Example: Traditional shell scripts or batch scripts where you list each command to run in sequence are imperative.

Declarative Configuration

- In a declarative approach, you specify what you want the system to look like, not how to achieve that state
- Concentrates on the desired end state
- The system or tool figures out the necessary steps to reach that state
- Often simpler and more readable, as you don't need to specify every step
- Reduces the chance of errors since the tool handles the process.
- Example: Ansible playbooks, Terraform configurations, and Kubernetes manifests are primarily declarative

Declarative vs Imperative

- Clarity vs. Control
  - Declarative configurations are often clearer and more concise, focusing on the "what"
  - Imperative configurations give more control by focusing on the "how"
- Tool Responsibility
  - In declarative configurations, the tool is responsible for figuring out how to achieve the desired state, reducing potential errors
  - In imperative configurations, the responsibility lies more with the developer or operator
- Flexibility
  - While declarative tools are designed for specific use cases (e.g., Ansible for configuration management), imperative approaches can be more flexible and can handle a wider range of tasks
- Learning Curve
  - Declarative tools might have a steeper initial learning curve as users need to understand the tool's conventions and capabilities
  - Imperative approaches, being more manual, might be more intuitive initially but can become complex as tasks grow

Use Ansible Modules Properly

- Commands like **`shell`** or **`command`** are not inherently idempotent
- If you must use them, ensure idempotency by adding conditions
- Most Ansible modules are designed to be idempotent
- Always prefer using a module over running raw commands
- For example, use the file module to manage files instead of raw shell or command tasks.

Test with Check Mode

- Run playbooks with **`--check`** (check mode) to see what changes would be made without actually applying them
- A truly idempotent task will not report changes on subsequent runs unless the system state has changed

Non-idempotent vs Idenpontent way

- Non-idempotent way

```
tasks:
  - name: Install nginx using shell (not idempotent)
    shell: apt-get install nginx
```

- Idempotent way

```
tasks:
  - name: Ensure nginx is installed (idempotent)
    apt:
      name: nginx
      state: present
```

Idempotent Playbook

Demo