Ansible

DevOps Advanced

Agenda

- What is Ansible?
- Ansible Architecture
- Running Ad-Hoc Commands
- Authentication
- Inventories
- Dynamic Inventories
- Playbooks

What is Ansible?

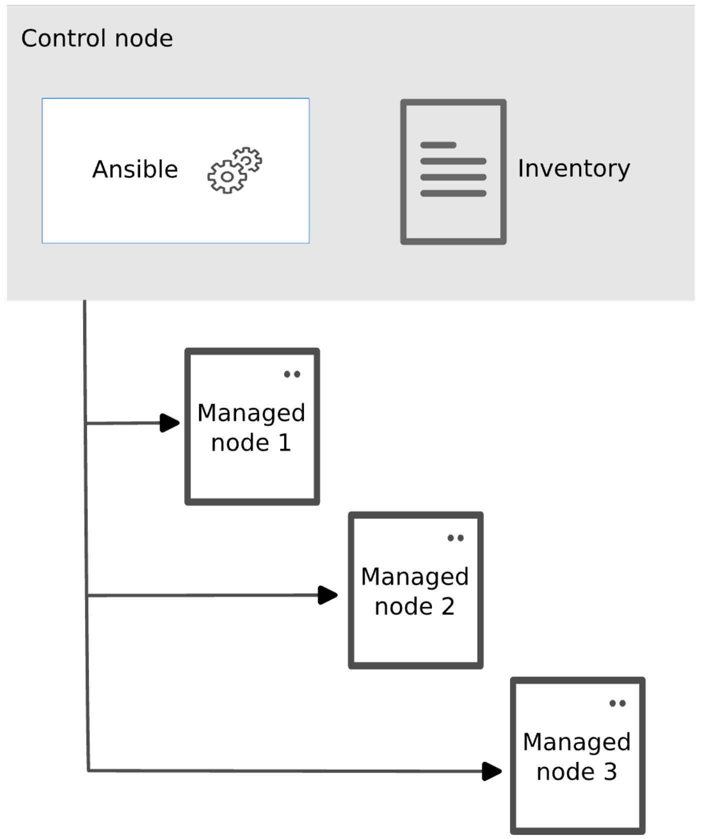
DevOps Advanced

What is Ansible?

- Open-Source Automation Tool: Ansible is a free and open-source tool designed for automation tasks such as configuration management, application deployment, and infrastructure provisioning.
- Agentless Design: Unlike many other automation tools, Ansible operates in an agentless manner, relying on SSH (or WinRM for Windows) to communicate with target machines, eliminating the need to install additional software on those machines.
- Playbook-Driven: Ansible uses playbooks written in YAML (Yet Another Markup Language) to define and describe automation tasks in a human-readable format.
- Extensibility: With a vast library of built-in modules and the ability to create custom ones, Ansible can manage and automate diverse systems, from servers and databases to networking devices and cloud resources.

What is Ansible?

- **Control node**: A system on which Ansible is installed. You run Ansible commands such on a control node.
- **Managed node**: A remote system, or host, that Ansible controls.
- **Inventory**: A list of managed nodes that are logically organized. You create an inventory on the control node to describe host deployments to Ansible.



What Ansible Automates

- Configuration Management
  - Ensure consistent configurations across servers.
  - Apply system updates and patches.
  - Manage users and groups.
- Application Deployment
  - Deploy applications to various environments.
  - Coordinate multi-tier rollouts.
  - Handle database migrations and updates.
- Infrastructure Provisioning
  - Spin up and tear down virtual machines or cloud instances.
  - Manage cloud resources, such as storage, networking, and databases.
  - Work with popular cloud providers like AWS, Azure, and Google Cloud.
- Orchestration
  - Coordinate tasks on multiple machines.
  - Handle complex deployments and workflows.
  - Manage rolling updates without downtime.

What Ansible Automates

- Network Automation
  - Configure network devices from vendors like Cisco, Juniper, and Arista.
  - Automate network setups, VLAN configurations, and firewall rules.
- Security and Compliance
  - Harden systems based on security benchmarks.
  - Ensure compliance with policies and standards.
  - Rotate passwords and manage SSH keys.
- Continuous Delivery and Integration
  - Integrate with CI/CD pipelines.
  - Automate testing and validation processes.
  - Handle artifact delivery and staging.
- Task Automation
  - Run ad-hoc commands across a group of servers.
  - Schedule routine tasks like backups and cleanups.
- Storage Management
  - Automate storage provisioning and management.
  - Handle tasks related to file systems, LVM, and RAID configurations.

What Ansible Automates

- Monitoring and Logging
  - Integrate with monitoring solutions.
  - Automate the setup of logging solutions and forwarders.
- Backup and Recovery
  - Automate backup processes for databases, applications, and systems.
  - Handle restoration processes.
- Cloud Services Integration
  - Manage services like AWS Lambda, S3, RDS, and more.
  - Automate scaling actions and resource adjustments.
- Custom Extensions
  - With its modular design, users can write custom modules to automate tasks specific to their needs.

Control Node Requirements

- For your control node (the machine that runs Ansible), you can use nearly any UNIX-like machine with Python 3.9 or newer installed
- This includes Red Hat, Debian, Ubuntu, macOS, BSDs, and Windows under a Windows Subsystem for Linux (WSL) distribution
- Windows without WSL is not natively supported as a control node; see (more details here)

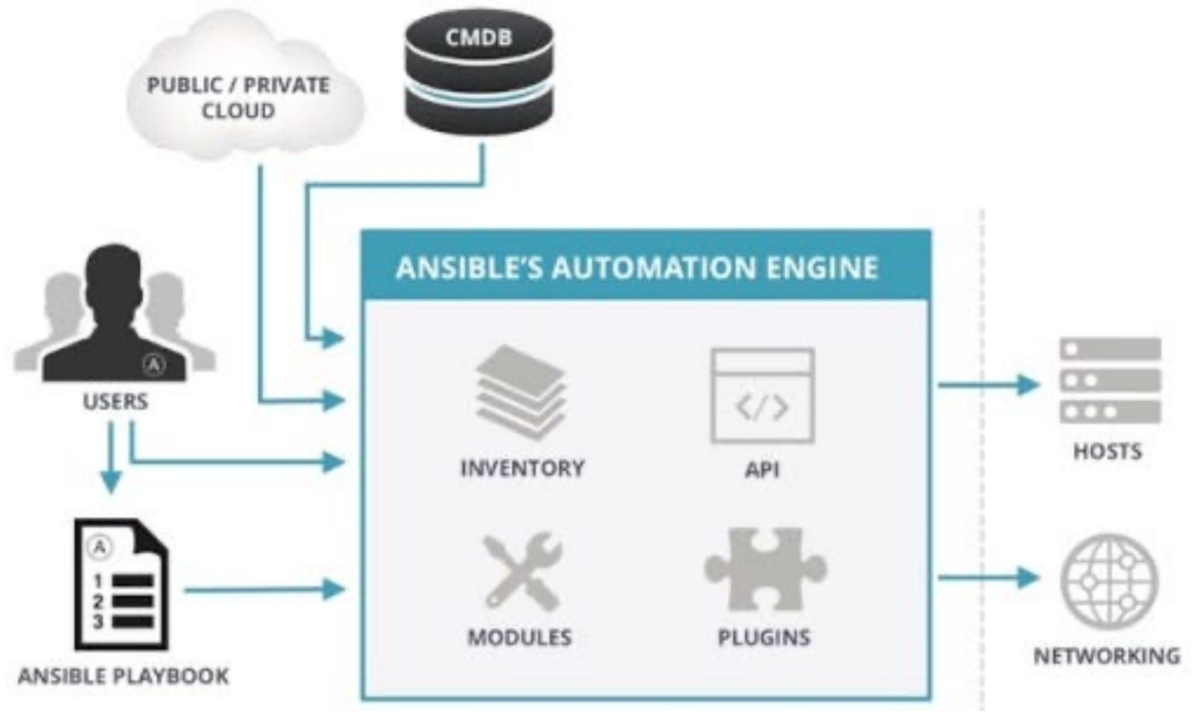
Managed Node Requirements

- The Managed node (the machine that Ansible is managing) does not require Ansible to be installed, but requires Python 2.7, or Python 3.5 - 3.11 to run Ansible-generated Python code
- The managed node also needs a user account that can connect through SSH to the node with an interactive POSIX shell.

Ansible Architecture

DevOps Advanced

Ansible Architecture

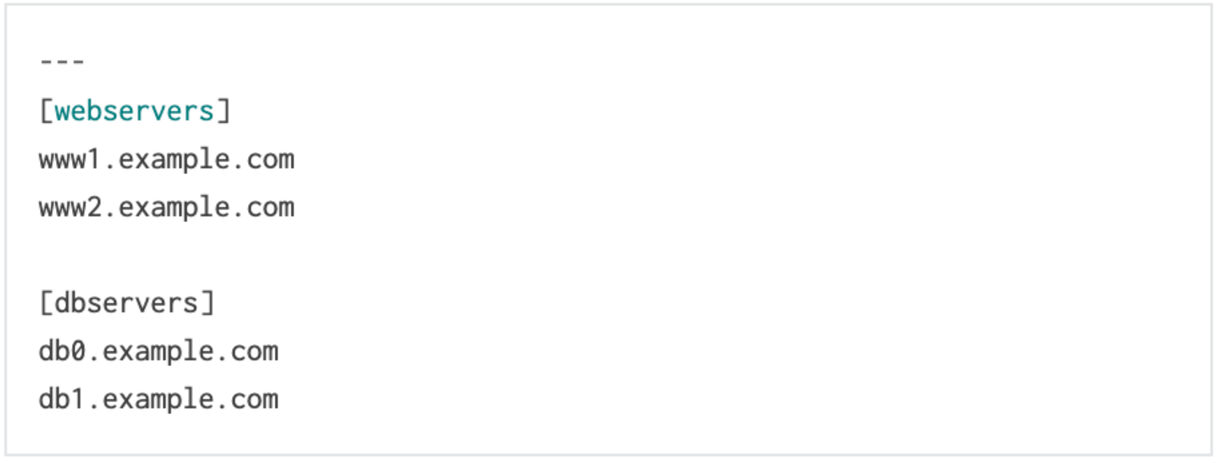


Automation Engine: Inventory

- By default, Ansible represents what machines it manages using a very simple INI file creating a static inventory, using groups of your own choosing
- Other Ansible option, is to use a dynamic inventory to pull your inventory from data sources like AWS, Azure, vSphere, etc.
- Once inventory hosts are listed, variables can be assigned to them in simple text files (in a subdirectory called ‘group_vars/’ or ‘host_vars/’ or directly in the inventory file.

Automation Engine: Inventory

- Example of a inventory file



```
---
[webservers]
www1.example.com
www2.example.com

[dbservers]
db0.example.com
db1.example.com
```

Automation Engine: Modules

- Building Blocks of Tasks: Modules are the fundamental units in Ansible used to perform individual tasks. When you define a task in a playbook, you are essentially calling a module with specific parameters.
- Wide Range of Functions: Ansible comes with a vast library of built-in modules that can manage system resources, software packages, files, cloud infrastructure, network devices, and much more.
- Idempotent Operations: Like playbooks, most modules are designed to be idempotent. This means that they will only make changes if the current state of the system doesn't match the desired state.
- Extensibility: While Ansible provides a plethora of built-in modules, users can also write custom modules to cater to specific needs or to manage unique systems.
- Language Agnostic: Although Ansible is written in Python, modules can be written in any language. As long as the language can return JSON, it can be used to write an Ansible module.

Automation Engine: Plugins

- Extensions to Core Features: Plugins augment Ansible's core functionality. They are small pieces of code that enhance built-in capabilities, allowing Ansible to interact with various systems, software, and APIs.
- Diverse Categories: There are many types of plugins in Ansible, including inventory, action, callback, connection, filter, lookup, and more, each serving a specific purpose.
- Customization: While Ansible comes with a wide range of built-in plugins, users can also develop custom plugins to meet specific needs or to integrate with unique systems.
- Execution: Unlike modules, which are executed on the target machine, plugins typically run on the control machine, enhancing the functionality of the Ansible core engine.
- Configuration: Plugins can be configured within the ansible.cfg file, allowing users to define specific settings or choose which plugins to use for specific tasks.

Automation Engine: Modules vs Plugins

- Ansible Modules
  - Purpose: Modules are the units of work in Ansible. They are the commands or set of similar commands executed on the target machine(s).
  - Execution Location: Modules are executed on the target machine(s). The control machine sends the module to the target, where it's executed.
  - Language: While most built-in modules are written in Python, they can technically be written in any language. The key is that they accept and return JSON.
  - Usage: Used within tasks in playbooks to perform actions, manage state, or gather data from target systems.
- Ansible Plugins
  - Purpose: Plugins augment or extend Ansible's core functionality. They modify or enhance the behavior of Ansible but don't directly perform tasks on target machines.
  - Execution Location: Plugins are executed on the control machine, not on the target systems.
  - Language: Plugins are primarily written in Python.
  - Usage: Enhance the functionality of playbooks, configurations, and runtime operations. They can be used to customize Ansible's behavior, integrate with external systems, or process data.

Automation Engine: Playbooks

- YAML Scripts: Playbooks are written in YAML (Yet Another Markup Language), which is both human-readable and machine-parsable, ensuring clarity in defining tasks.
- Ordered Set of Tasks: Playbooks define an ordered set of tasks to be executed against host machines, ensuring configurations are applied in the correct sequence.
- Idempotent Operations: Tasks in playbooks are designed to be idempotent, meaning they can be executed multiple times without changing the result after the first successful run.
- Multi-Tier Orchestration: Playbooks can manage configurations across multi-tier setups, allowing for complex deployments involving databases, application servers, web servers, etc.
- Variables & Templates: Playbooks support the use of variables and templates, enabling dynamic configurations based on specific conditions or inputs.

Run ad-hoc commands

DevOps Advanced

Ad-Hoc Commands

- An ad hoc command uses the **`/usr/bin/ansible`** command-line tool to automate a single task on one or more managed nodes.
- Ad hoc commands are quick and easy, but they are not reusable.
- Ad hoc commands demonstrate the simplicity and power of Ansible
- The concepts you learn here will port over directly to the playbook language. Before reading and executing these examples, please read

Why Using Ad-hoc Commands?

- Ad hoc commands are great for tasks you repeat rarely
- For example, if you want to power off all the machines in your lab for Christmas vacation, you could execute a quick one-liner in Ansible without writing a playbook
- An ad hoc command looks like this:

```
$ ansible [pattern] -m [module] -a "[module options]"
```

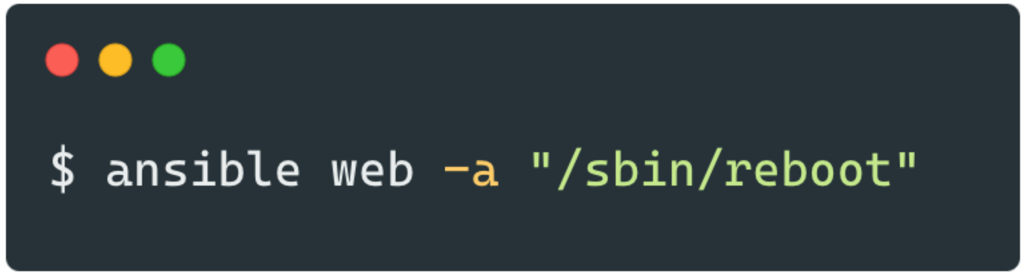
- The **`-a`** option accepts options either through the **`key=value`** syntax or a JSON string starting with { and ending with } for more complex option structure

How to identify machines

- Ad hoc commands uses inventories to define the machines to be reached
- Inventory may be referenced in different ways following this precedence list
  - Command Line **`-i`** Option
  - **`ANSIBLE_INVENTORY`** Environment Variable
  - **`ansible.cfg`** Configuration File:
  - Default Inventory Path: **`/etc/ansible/hosts`**
- Inventory file defines groups of hosts and identify them with a name
- When executing the command you use patterns to define which machines you will run the command

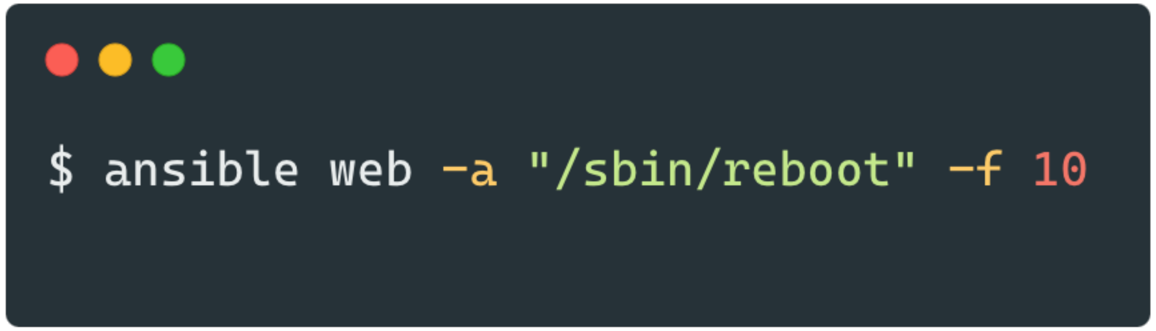
Simple ad-hoc commands

- Reboot all servers on **`web`** group on inventory



```
$ ansible web -a "/sbin/reboot"
```

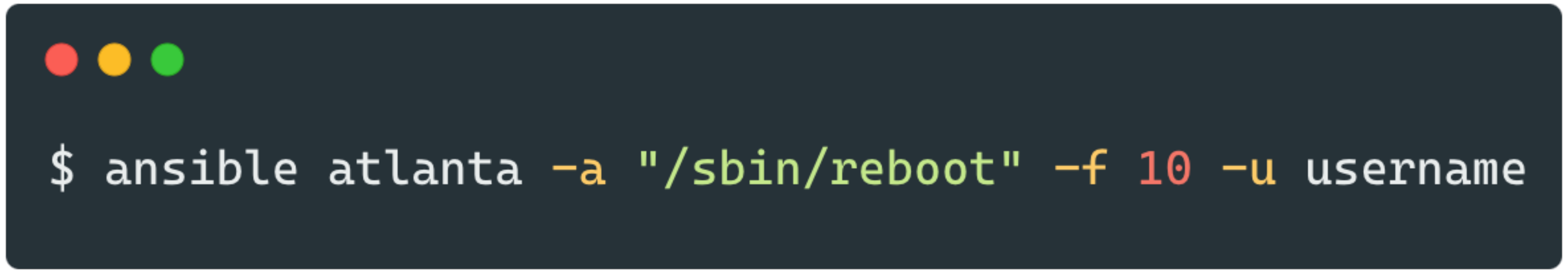
- Ansible runs 5 execution simultaneous. If you want to execute 10, you use flag **`-f`**



```
$ ansible web -a "/sbin/reboot" -f 10
```

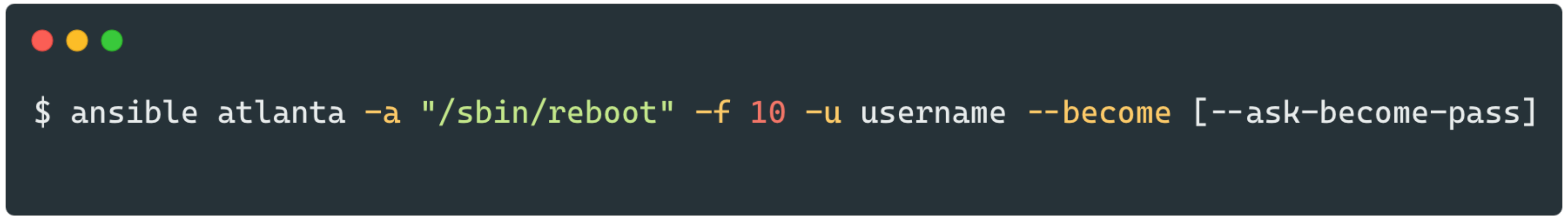
Simple ad-hoc commands

- If you want to execute the command with another username, you may use **-u** flag



```
$ ansible atlanta -a "/sbin/reboot" -f 10 -u username
```

- And sometimes you need to elevate your user



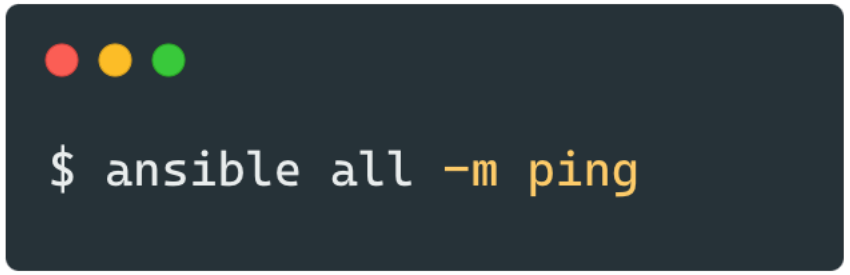
```
$ ansible atlanta -a "/sbin/reboot" -f 10 -u username --become [--ask-become-pass]
```

Run ad-hoc commands using modules

- On previous samples, you're using the **`ansible.builtin.command`**
- This is the default module and you don't need to specify on every command
- The flag **`–a`** represent the argument for module command, that on this case represent the command itself
- To specify a different module you may use **`–m`** flag
- Module name can be composed by several blocks using a dot to separate them
- When you do use this full name, the module will be related with **`ansible.builtin`**

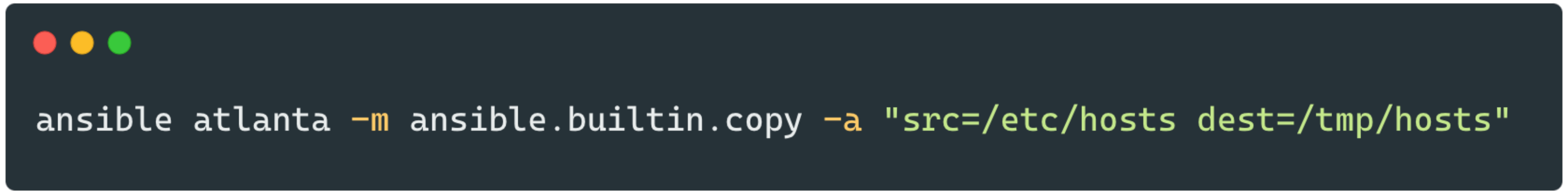
Run ad-hoc commands using modules

- Ping all machines



```
$ ansible all -m ping
```

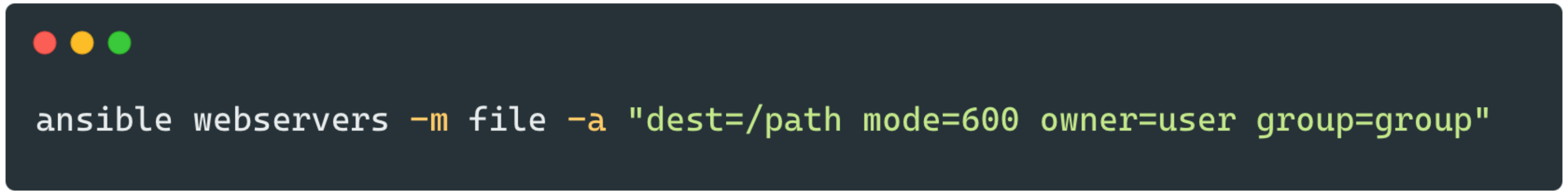
- Copy **`/etc/hosts`** to **`/tmp/hosts`** on atlanta group within the inventory



```
ansible atlanta -m ansible.builtin.copy -a "src=/etc/hosts dest=/tmp/hosts"
```

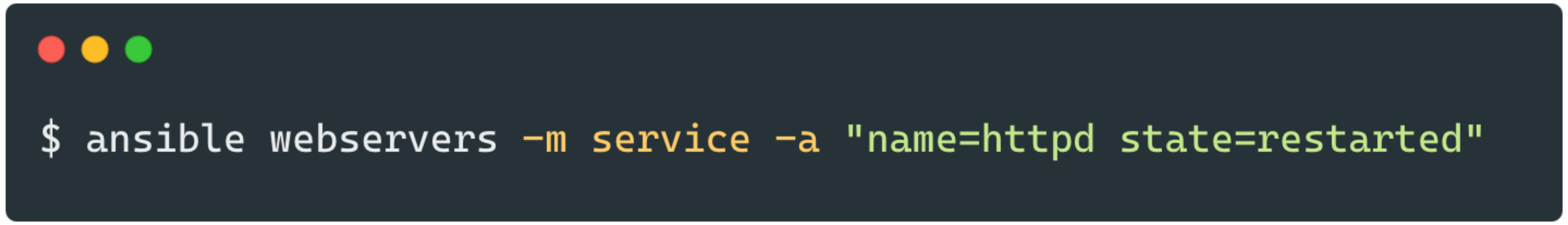
Run ad-hoc commands using modules

- You may create folders with several parameters



```
ansible webservers -m file -a "dest=/path mode=600 owner=user group=group"
```

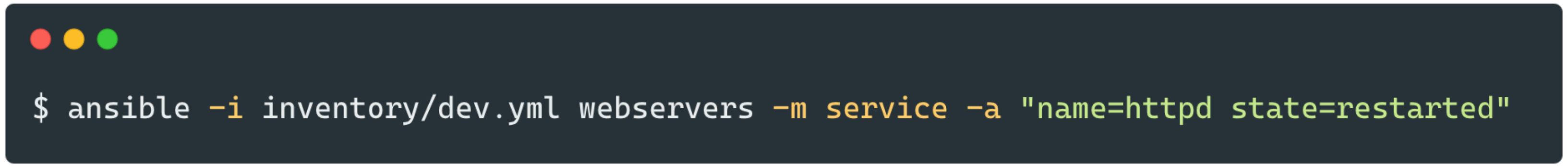
- Restart **`httpd`** service on **`webservers`** group within the inventory



```
$ ansible webservers -m service -a "name=httpd state=restarted"
```

Run ad-hoc commands with explicit inventory

- Using **`-i`** flag to define inventory



```
$ ansible -i inventory/dev.yml webservers -m service -a "name=httpd state=restarted"
```

Modules names

- In modern Ansible practices, it's recommended to use the fully qualified collection name (FQCN) for modules
- This practice helps in clearly distinguishing between core modules and those from other collections, ensuring clarity and avoiding conflicts.
- For example, instead of using just copy, ping, or file, you would use:
  - **`ansible.builtin.copy`**
  - **`ansible.builtin.ping`**
  - **`ansible.builtin.file`**
- This is especially useful in environments where custom modules or modules from other collections are also being used.

Authenticating Ansible Connections

Introduction to Ansible

How to authenticate

- SSH Keys
  - Ansible primarily uses SSH keys for authentication
  - It's the most common method where the control machine has a private key and the managed nodes have the corresponding public key in the authorized keys list
- Username & Password
  - While less secure and not recommended for production, Ansible can use SSH with a username and password for authentication
  - This method can be useful in scenarios where key-based authentication isn't feasible
  - You need to have **`sshpass`** installed on control node

SSH Key

- Ansible leverages the native SSH mechanism for authentication, which means it uses the SSH keys already defined on the control node (the machine running Ansible) to authenticate to remote servers
- Default Private Key
  - By default, Ansible uses the private key located at **`~/.ssh/id_rsa`**
- Specifying a Different Key
  - If you want to use a different private key, you can specify it using the **`--private-key`** option or set it in the Ansible configuration file (**`ansible.cfg`**) using the **`private_key_file`** setting.
- ssh-agent
  - If you're using **`ssh-agent`** on the control node, Ansible can leverage it.
  - When you add your private keys to the agent using **`ssh-add`**, Ansible will use the identities loaded into the agent for authentication.
- Ansible Vault
  - If you need to store SSH private keys securely, you can encrypt them using Ansible Vault.
  - When running playbooks, you can decrypt the key on-the-fly, ensuring that sensitive keys are not exposed in plaintext.

Run Ad-Hoc Commands

Demo

Inventory

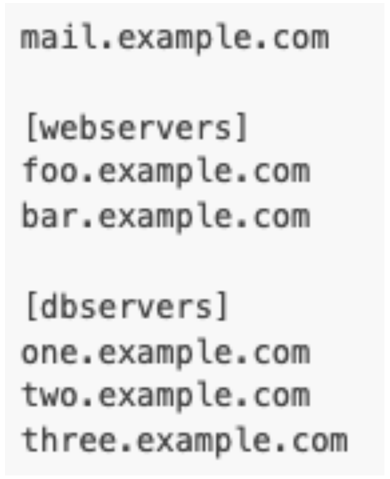
DevOps Advanced

What is inventory?

- Inventory is a list of machines that can be used on your ansible commands
- Inside inventory you must define groups that allow you to identify your servers on a easy way
- Inventory may be referenced in different ways following this precedence list
  - Command Line **`-i`** Option
  - **`ANSIBLE_INVENTORY`** Environment Variable
  - **`ansible.cfg`** Configuration File:
  - Default Inventory Path: **`/etc/ansible/hosts`**
- Some best practices: Ansible Best Practices

Inventory Format

- You may use two formats: INI and YAML

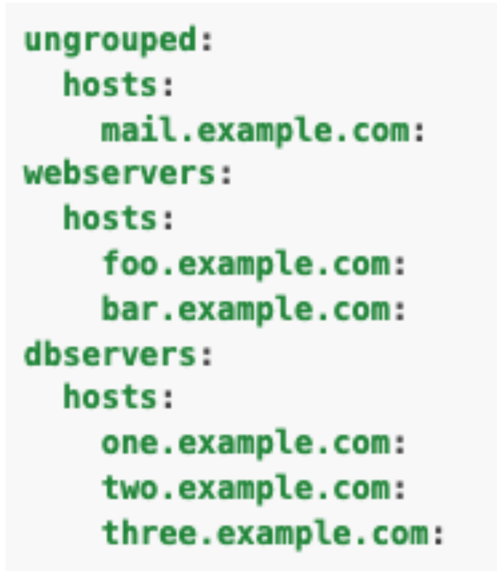


```
mail.example.com

[webservers]
foo.example.com
bar.example.com

[dbservers]
one.example.com
two.example.com
three.example.com
```

INI



```yaml
ungrouped:
  hosts:
    mail.example.com:
webservers:
  hosts:
    foo.example.com:
    bar.example.com:
dbservers:
  hosts:
    one.example.com:
    two.example.com:
    three.example.com:
```

YAML

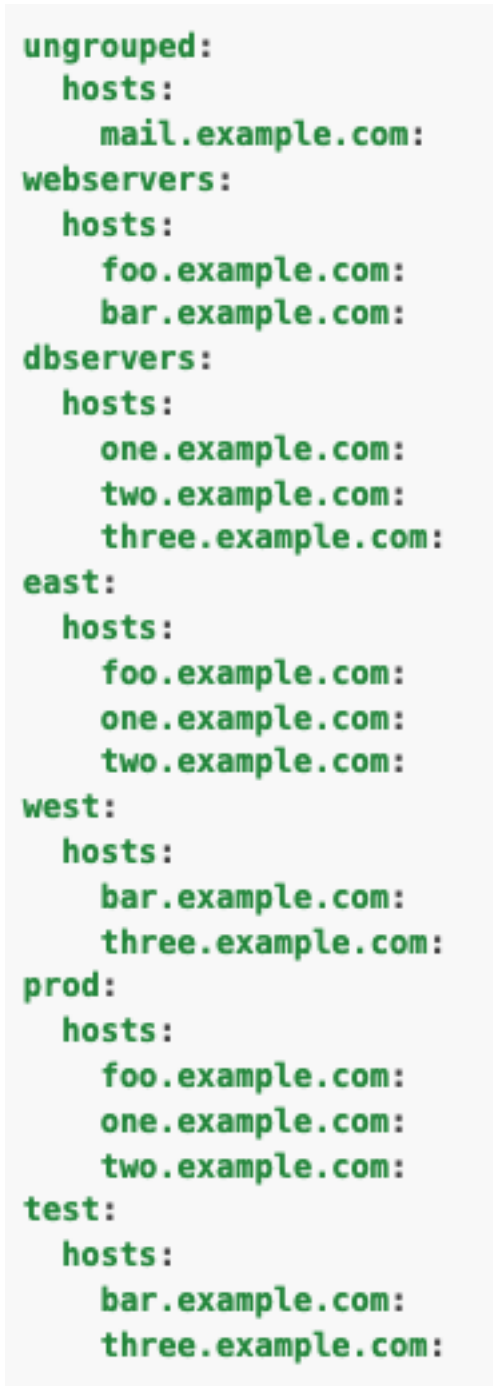
Default groups

- Even if you do not define any groups in your inventory file, Ansible creates two default groups: all and ungrouped
- The all group contains every host
- The ungrouped group contains all hosts that don’t have another group aside from all
- Every host will always belong to at least 2 groups (all and ungrouped or all and some other group)

Hosts in multiple groups

- You can (and probably will) put each host in more than one group.
- For example a production webserver in a datacenter in Atlanta might be included in groups called `[prod]` and `[atlanta]` and `[webservers]`.
- You can create groups that track:
  - What - An application, stack or microservice (for example, database servers, web servers, and so on).
  - Where - A datacenter or region, to talk to local DNS, storage, and so on (for example, east, west).
  - When - The development stage, to avoid testing on production resources (for example, prod, test).

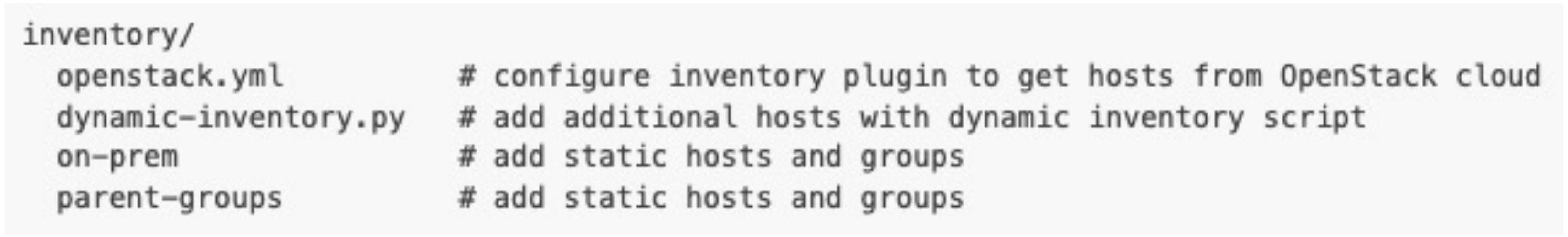
Hosts in multiple groups



```
ungrouped:
  hosts:
    mail.example.com:
webservers:
  hosts:
    foo.example.com:
    bar.example.com:
dbservers:
  hosts:
    one.example.com:
    two.example.com:
    three.example.com:
east:
  hosts:
    foo.example.com:
    one.example.com:
    two.example.com:
west:
  hosts:
    bar.example.com:
    three.example.com:
prod:
  hosts:
    foo.example.com:
    one.example.com:
    two.example.com:
test:
  hosts:
    bar.example.com:
    three.example.com:
```

Organizing inventory in a directory

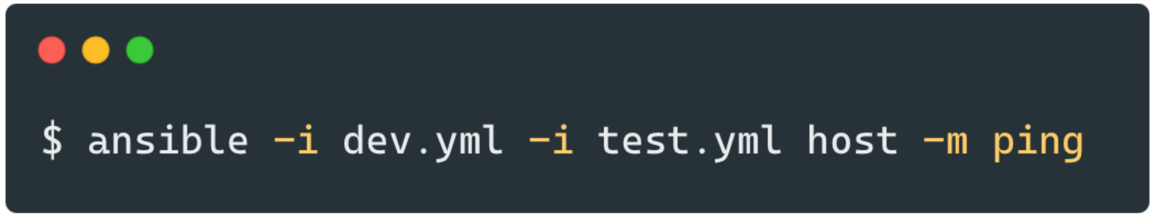
- You can consolidate multiple inventory sources in a single director
- Simplest version of this is a directory with multiple files instead of a single inventory file
- The following inventory directory combines an inventory plugin source, a dynamic inventory script, and a file with static hosts:



```
inventory/
  openstack.yml          # configure inventory plugin to get hosts from OpenStack cloud
  dynamic-inventory.py   # add additional hosts with dynamic inventory script
  on-prem                # add static hosts and groups
  parent-groups          # add static hosts and groups
```

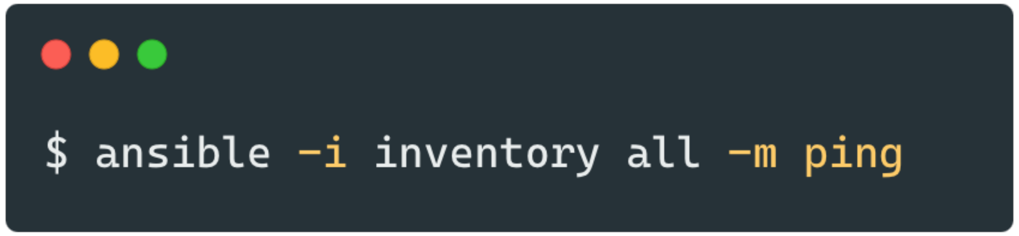
Passing multiple inventory sources

- You can target multiple inventory at the same time by giving multiple inventory parameters from the command line



```
$ ansible -i dev.yml -i test.yml host -m ping
```

- Or you can target this inventory directory as follows:



```
$ ansible -i inventory all -m ping
```

Dynamic Inventories

Inventories

Dynamic Inventories

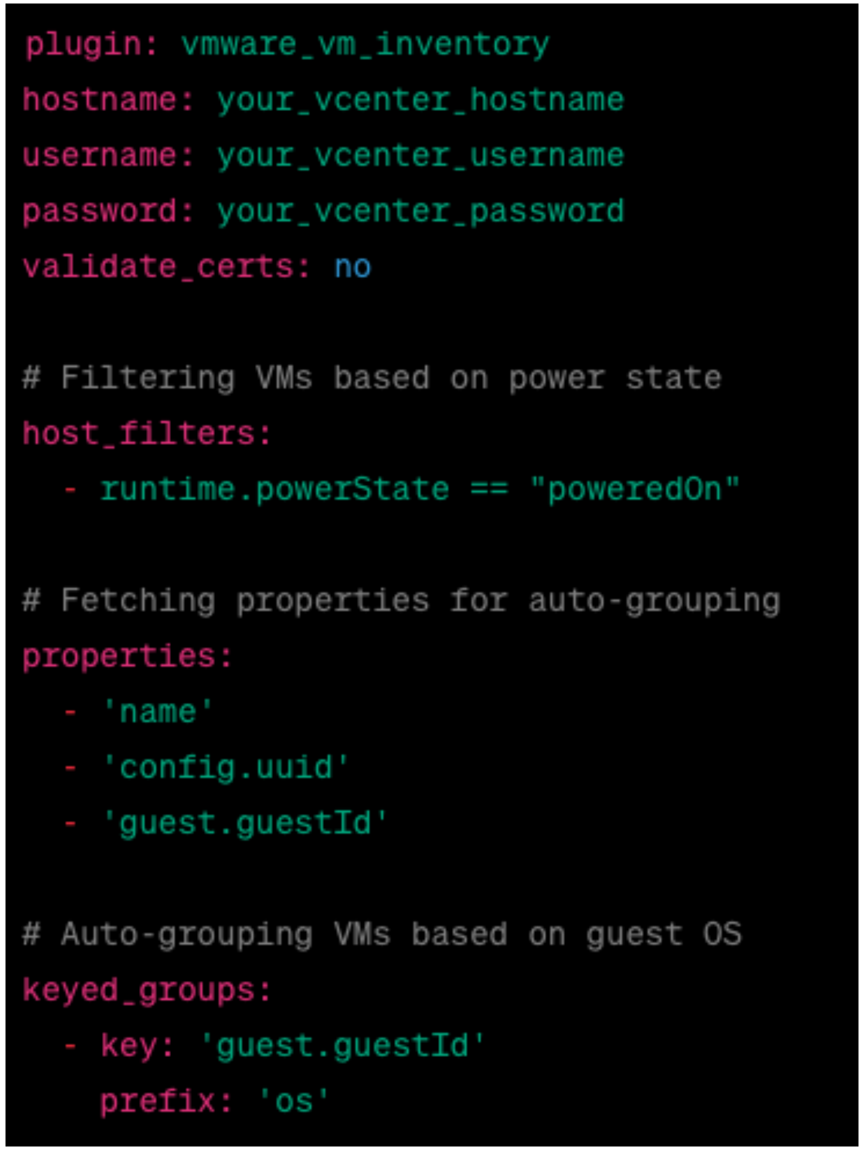
- Definition: Dynamic inventories provide a way for Ansible to fetch a list of hosts from external sources at runtime, rather than relying on a statically defined inventory file.
- Flexibility: They are especially useful in cloud environments where infrastructure can be ephemeral, with hosts being created and destroyed dynamically.
- Scripts & Plugins: Dynamic inventories can be implemented using scripts (legacy method) or plugins (recommended). These scripts or plugins fetch host information and output it in a JSON format that Ansible understands.
- Grouping: Dynamic inventory plugins can automatically group hosts based on common attributes, such as tags in cloud environments.

Dynamic Inventories

- Caching: To improve performance, Ansible can cache the results of dynamic inventories, ensuring faster execution times.
- Configuration: Dynamic inventories often require configuration files where credentials, filters, and other settings specific to the external source are defined.
- Multiple Sources: You can use multiple dynamic inventory sources together, allowing for a mix of static and dynamic hosts.
- Built-in Support: Ansible has built-in support for many popular cloud providers and platforms for dynamic inventories, including AWS, Azure, VMware, and more.

Dynamic Inventories: VMWare

- Needs a Python plugin called **`pyvmomi`** (https://github.com/vmware/pyvmomi)
- Then you can create your inventory file using VMWare credentials and some host filtering



```yaml
plugin: vmware_vm_inventory
hostname: your_vcenter_hostname
username: your_vcenter_username
password: your_vcenter_password
validate_certs: no

# Filtering VMs based on power state
host_filters:
  - runtime.powerState == "poweredOn"

# Fetching properties for auto-grouping
properties:
  - 'name'
  - 'config.uuid'
  - 'guest.guestId'

# Auto-grouping VMs based on guest OS
keyed_groups:
  - key: 'guest.guestId'
    prefix: 'os'
```

Static vs Dynamic Inventories

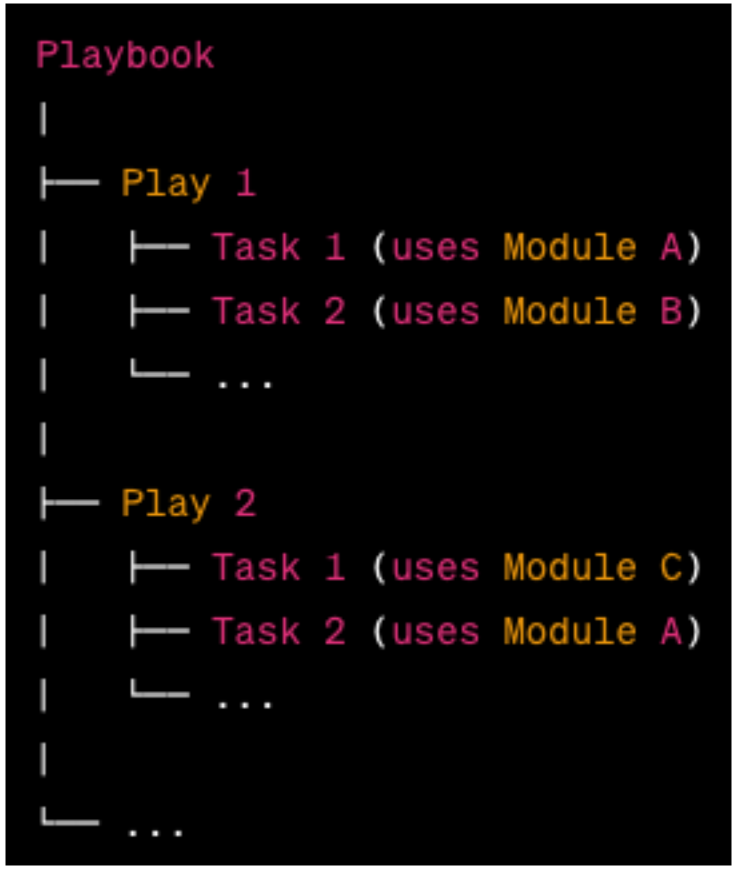
Type	Pros	Cons
Static	Simplicity Stability No Dependencies Fine-grained Control	Manual Updates Scalability Issues Duplication
Dynamic	Automation Scalability Flexibility Auto Grouping	Complexity Dependencies Potential Delays Security Concerns

Playbook

DevOps Advanced

Modules, Tasks, Plays, Playbooks

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



```
Playbook
|
├── Play 1
|   ├── Task 1 (uses Module A)
|   ├── Task 2 (uses Module B)
|   └── ...
|
├── Play 2
|   ├── Task 1 (uses Module C)
|   ├── Task 2 (uses Module A)
|   └── ...
|
└── ...
```

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

Play

- A play is a set of tasks that will be run on a particular set of hosts in a sequence
- 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.

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.

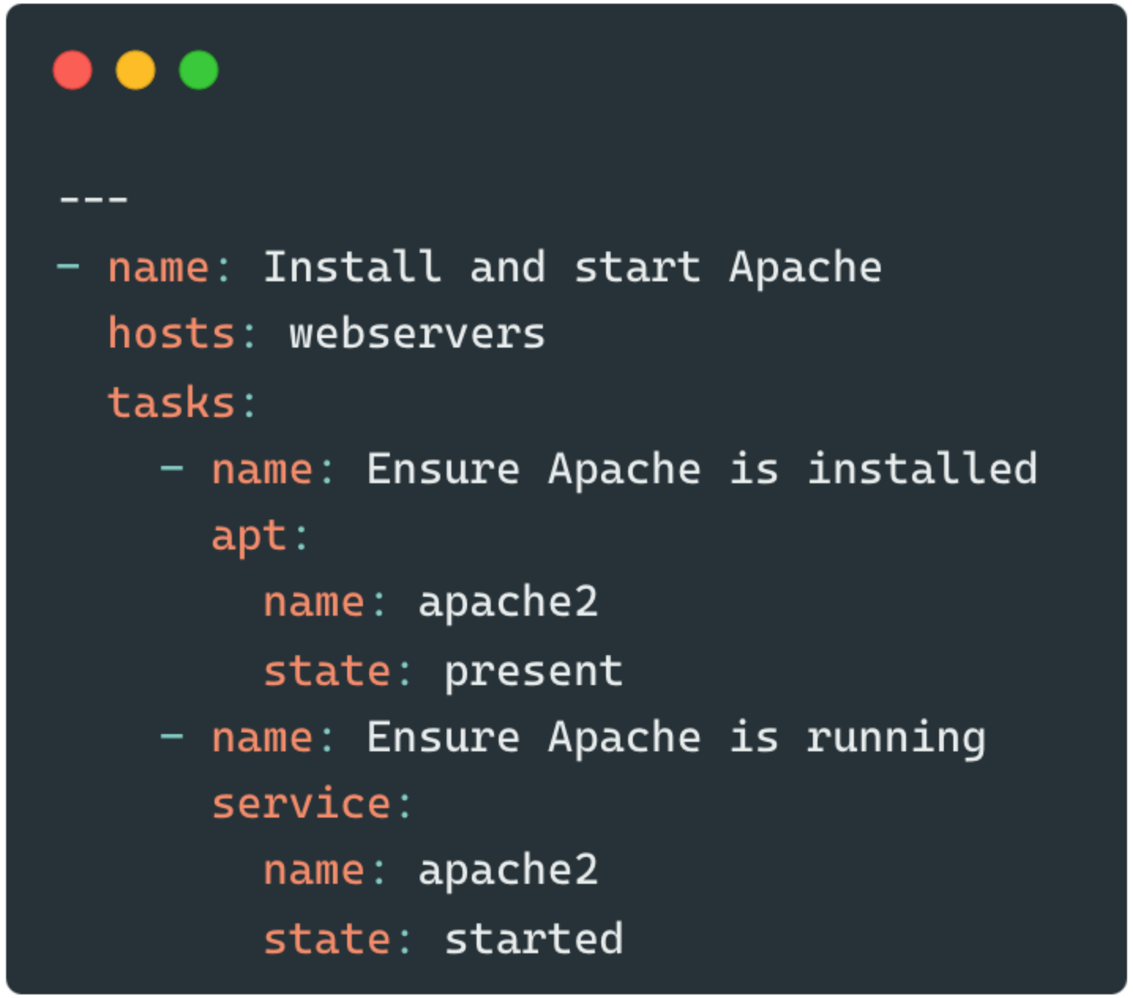
Modules

- 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.

Basic Play Structure

- `name`
  - Specify play name
  - Important for identify on logs
- `hosts`
  - Specifies which hosts the tasks will run on
  - Can target individual hosts, groups, or patterns
- `tasks`
  - A list of tasks to execute in order
  - Each task calls an Ansible module.

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
```

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
    - Inside one task, several hosts run on parallel

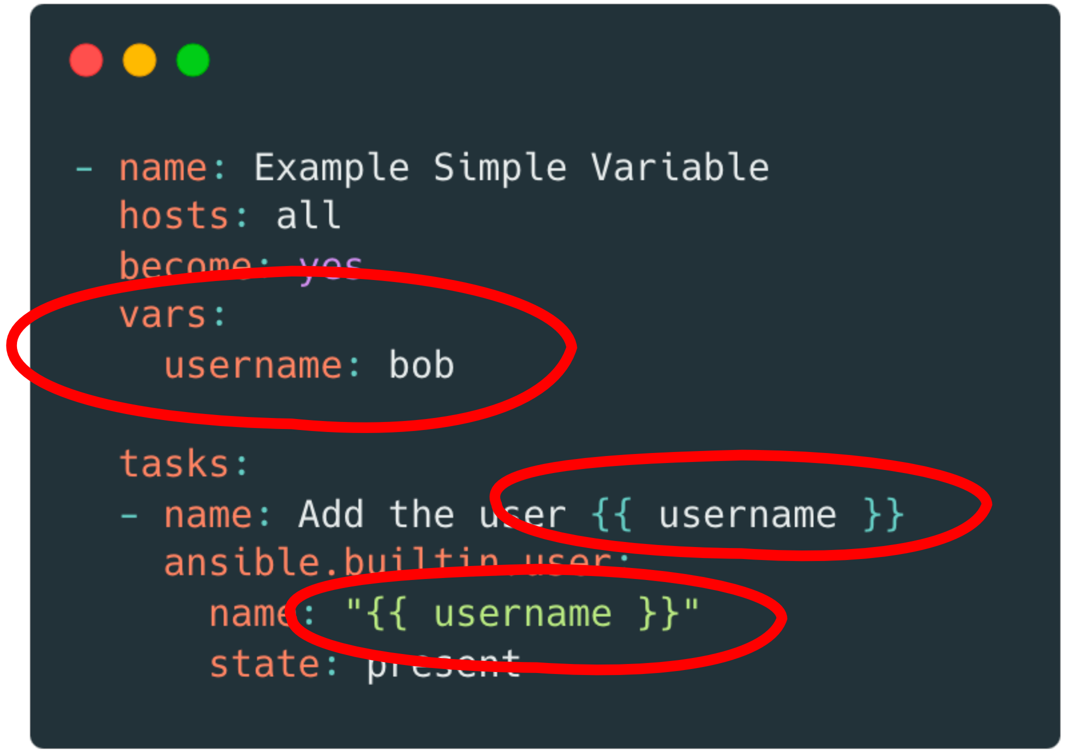
Variables

- You can define them directly on the playbook, using group variables or host variables
- To reference a variable you may user the format **`{{ var_name }}`**
- 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

Playbook with Variables

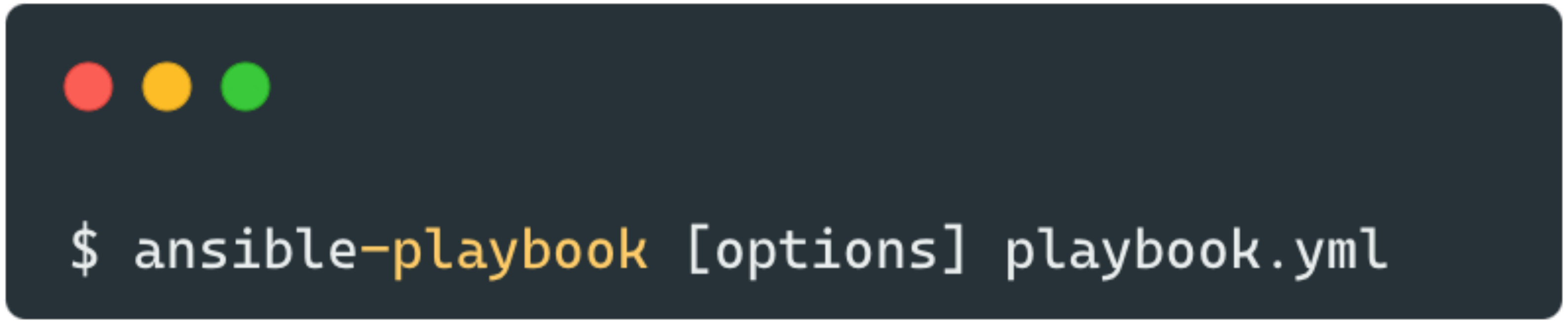


```yaml
- name: Example Simple Variable
  hosts: all
  become: yes
  vars:
    username: bob

  tasks:
  - name: Add the user {{ username }}
    ansible.builtin.user:
      name: "{{ username }}"
      state: present
```

Execute Playbook

- Using ansible-playbook command



```
$ ansible-playbook [options] playbook.yml
```

Run your First Playbook

Demo

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

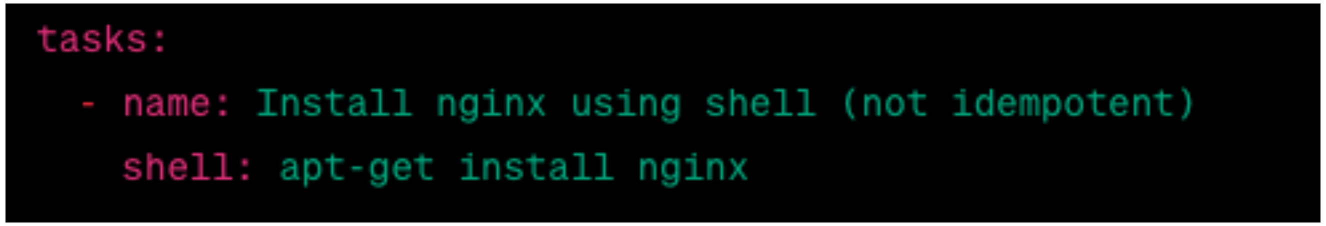
- Clarity vs. Control
  - Declarative configurations are often clearer and more concise, focusing on the "what”
  - Imperative configurations give more control by focusing on the "how"
- Tool Responsibility
  - In declarative configurations, the tool is responsible for figuring out how to achieve the 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.

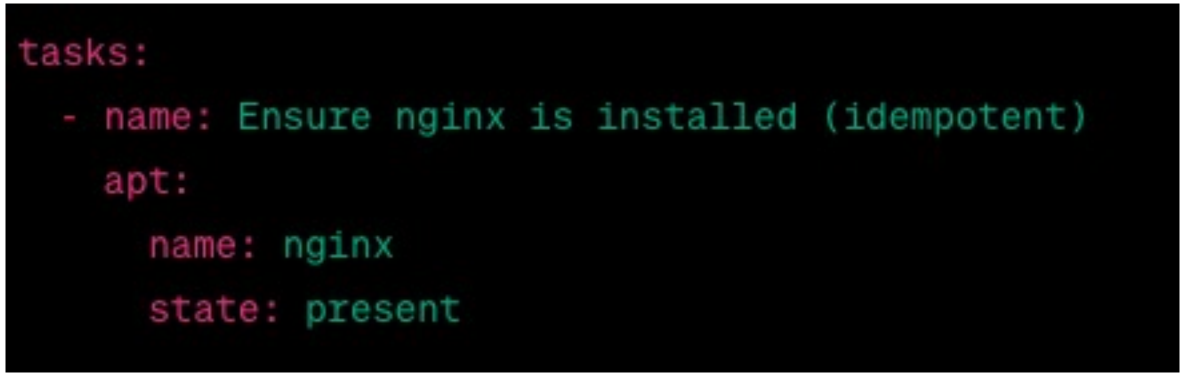
Non-idempotent vs Idenpontent way

- 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
```

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

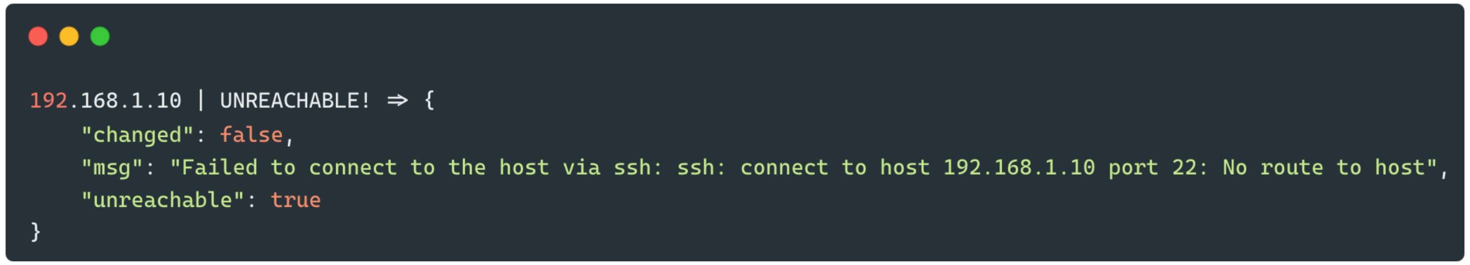
- The task did not execute successfully and encountered an error.
- An error occurred that prevented the module from completing its operation.
- This could be due to various reasons like incorrect parameters, issues on the target system, unreachable hosts, etc.
- Example: If you have a task to ensure 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"

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
}
```

Check Task Results

Demo