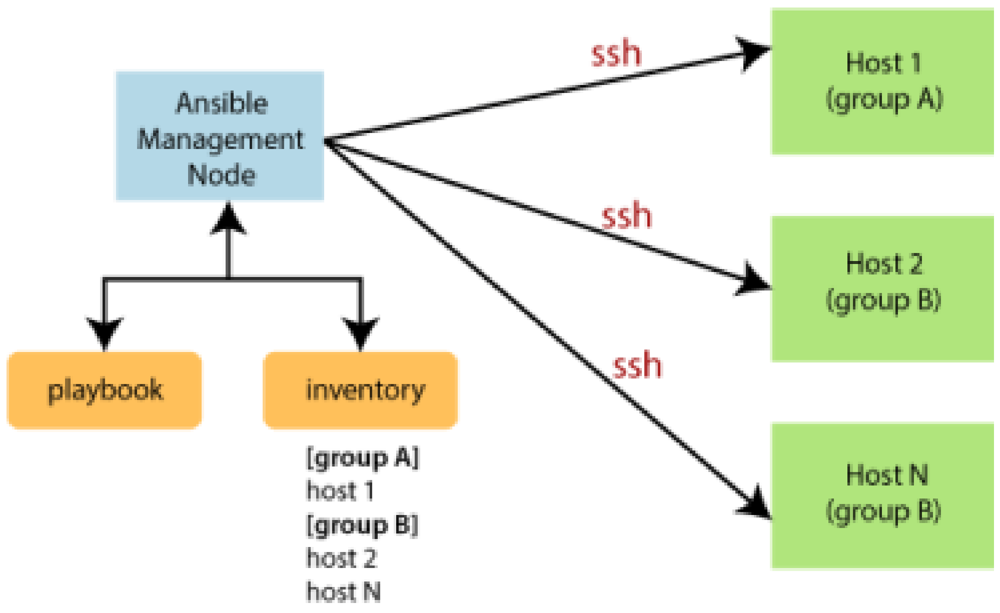
ANSIBLE

■ Ansible is an open-source IT-Configuration Management, Deployment, and Orchestration tool. It aims to provide large productivity gains to a wide variety of automation challenges.

■ Ansible Architecture



Ansible Notes 1

Ansible Architecture is too simple just one Management Node and others will be host in where we want our configuration management through an ssh connection.

█ Terms used in Ansible

Ansible Server → The machine where Ansible is installed and from which all tasks and playbooks will be run.

Module → Module is a command or set of similar commands meant to be executed on the client side.

Task → A task is a section that consists of a single procedure to be completed.

Role → A way of organizing tasks and related files to be later called in a playbook.

Fact → Information fetched from the client system from the global variables with the gather facts operation.

Inventory → File containing data about the Ansible client server.

Play → Execution of a playbook.

Handler → Task which is called only if a notifier is present.

Notifier → Section attributed to a task that calls a handler if the output is changed.

Playbook → It consists of YAML format code describing tasks to be executed.

Host → Nodes, which are automated by Ansible.

How does Ansible work?

Ansible uses the concepts of control and managed nodes. It connects from the **control node**, any machine with Ansible installed, to the **managed nodes** sending commands and instructions to them.

The units of code that Ansible executes on the managed nodes are called **modules**. Each module is invoked by a **task**, and an ordered list of tasks together forms a **playbook.** Users write playbooks with tasks and modules to define the desired state of the system.

The managed machines are represented in a simplistic **inventory** file that groups all the nodes into different categories.

Ansible leverages a very simple language, YAML, to define playbooks in a human readable data format that is really easy to understand from day one.

Even more, Ansible doesn't require the installation of any extra agents on the managed nodes so it's simple to start using it.

Typically, the only thing a user needs is a terminal to execute Ansible commands and a text editor to define the configuration files.

How to install Ansible

To start using Ansible, you will need to install it on a control node, this could be your laptop for example. From this control node, Ansible will connect and manage other machines and orchestrate different tasks.

The exact installation procedure depends on your machine and operating system but the most common way would be to use **pip**.

To install pip, in case it's not already available on your system:

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python get-pip.py --user
```

Ansible is one of the most used tools for managing cloud and on-premises infrastructure. If you are looking for a flexible and powerful tool to automate your infrastructure management and configuration tasks Ansible is the way to go.

In this introductory guide, you will learn everything you need to get started with Ansible and start building robust automation solutions.

What is Ansible?

Ansible is a software tool that enables cross-platform automation and orchestration at scale and has become over the years the standard choice among enterprise automation solutions.

It's mostly addressed to IT operators, administrators & decision-makers helping them to achieve operational excellence across their entire infrastructure ecosystem.

Backed by RedHat and a loyal open source community, it is considered an excellent option for configuration management, infrastructure provisioning, and application deployment use cases.

Its automation opportunities are endless across hybrid clouds, on-prem infrastructure, and IoT and it's an engine that can greatly improve the efficiency and

consistency of your IT environments.

Ready to automate everything? Let’s go!

How does Ansible work?

Ansible uses the concepts of control and managed nodes. It connects from the **control node**, any machine with Ansible installed, to the **managed nodes** sending commands and instructions to them.

The units of code that Ansible executes on the managed nodes are called **modules**. Each module is invoked by a **task**, and an ordered list of tasks together forms a **playbook.** Users write playbooks with tasks and modules to define the desired state of the system.

The managed machines are represented in a simplistic **inventory** file that groups all the nodes into different categories.

Ansible leverages a very simple language, YAML, to define playbooks in a human readable data format that is really easy to understand from day one.

Even more, Ansible doesn’t require the installation of any extra agents on the managed nodes so it’s simple to start using it.

Typically, the only thing a user needs is a terminal to execute Ansible commands and a text editor to define the configuration files.

Benefits of using Ansible

A free and open-source community project with a huge audience.

Battle-tested over many years as the preferred tool of IT wizards.

Easy to start and use from day one, without the need for any special coding skills.

Simple deployment workflow without any extra agents.

Includes some sophisticated features around modularity and reusability that come in handy as users become more proficient.

Extensive and comprehensive official documentation that is complemented by a plethora of online material produced by its community.

To sum up, Ansible is simple yet powerful, agentless, community-powered, predictable, and secure.

Basic Concepts & Terms

Host: A remote machine managed by Ansible.

Group: Several hosts grouped together that share a common attribute.

Inventory: A collection of all the hosts and groups that Ansible manages. Could be a static file in the simple cases or we can pull the inventory from remote sources, such as cloud providers.

Modules: Units of code that Ansible sends to the remote nodes for execution.

Tasks: Units of action that combine a module and its arguments along with some other parameters.

Playbooks: An ordered list of tasks along with its necessary parameters that define a recipe to configure a system.

Roles: Redistributable units of organization that allow users to share automation code easier. Learn why Roles are useful in Ansible.

YAML: A popular and simple data format that is very clean and understandable by humans.

How to install Ansible

To start using Ansible, you will need to install it on a control node, this could be your laptop for example. From this control node, Ansible will connect and manage other machines and orchestrate different tasks.

Installation Requirements

The exact installation procedure depends on your machine and operating system but the most common way would be to use **pip**.

To install pip, in case it's not already available on your system:

```
$ curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
$ python get-pip.py --user
```

After pip is installed:

```
$ python -m pip install --user ansible
```

You can test on your terminal if it's successfully installed by running:

```
$ ansible --version
```

Ansible ad hoc commands

Using ad hoc commands is a quick way to run a single task on one or more managed nodes.

Some examples of valid use cases are rebooting servers, copying files, checking connection status, managing packages, gathering facts, etc.

The pattern for ad hoc commands looks like this:

```
$ ansible [host-pattern] -m [module] -a “[module options]”
```

host-pattern: the managed hosts to run against

m: the module to run

a: the list of arguments required by the module

This is a good opportunity to use our first Ansible ad hoc command and at the same time validate that our inventory is configured as expected. Let’s go ahead and execute a ping command against all our hosts:

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

host1 | SUCCESS => {
        "ansible_facts": {
                "discovered_interpreter_python": "/usr/bin/python"
        },
        "changed": false,
        "ping": "pong"
}

host2 | SUCCESS => {
        "ansible_facts": {
                "discovered_interpreter_python": "/usr/bin/python"
        },
        "changed": false,
        "ping": "pong"
}
```

Nice, seems like we can successfully ping the 2 hosts that we have defined in our hosts file.

Next, run a live command only to the host2 node by using the **–limit** flag

```
$ ansible all -i hosts --limit host2 -a "/bin/echo hello"

host2 | CHANGED | rc=0 >>
hello
```

Intro to Ansible Playbooks

Playbooks are the simplest way in Ansible to automate repeating tasks in the form of reusable and consistent configuration files. Playbooks are defined in YAML files and contain any ordered set of steps to be executed on our managed nodes.

As mentioned, tasks in a playbook are executed from top to bottom. At a minimum, a playbook should define the managed nodes to target and some tasks to run against them.

In playbooks, data elements at the same level must share the same indentation while items that are children of other items must be indented more than their parents.

Let's look at a simple playbook to get an idea of how that looks in practice.

For the needs of this demo, we will use a simple playbook that runs against all hosts and copies a file, creates a user, and upgrades all apt packages on the remote machines.

```
---
- name: Intro to Ansible Playbooks
  hosts: all

  tasks:
  - name: Copy file hosts with permissions
    ansible.builtin.copy:
      src: ./hosts
      dest: /tmp/hosts_backup
      mode: '0644'
  - name: Add the user 'shubham'
    ansible.builtin.user:
      name: shubham
    become: yes
    become_method: sudo
  - name: Upgrade all apt packages
    apt:
      force_apt_get: yes
      upgrade: dist
    become: yes
```

On the top section, we define the group of hosts on which to run the playbook and its name. After that, we define a list of tasks. Each of the tasks contains some information about the task and the module to be executed along with the necessary arguments.

To avoid specifying the location of our inventory file every time we can define this via a configuration file (**ansible.cfg**).

```
[defaults]
inventory=./hosts
```

Using Variables in Playbooks

Variables can be defined in Ansible at more than one level and Ansible chooses the variable to use based on variable precedence.

Let's see how we can use variables at the playbook level.

The most common method is to use a **vars** block at the beginning of each playbook. After declaring them, we can use them in tasks. Use

{{ variable_name }} to reference a variable in a task.

```
---
- name: Variables playbook
  hosts: all
  vars:
      state: latest
      user: shubham
  tasks:
  - name: Add the user {{ user }}
    ansible.builtin.user:
      name: "{{ user }}"
  - name: Upgrade all apt packages
    apt:
      force_apt_get: yes
      upgrade: dist
  - name: Install the {{ state }} of package "nginx"
    apt:
      name: "nginx"
      state: "{{ state }}"
```

In the above example, we have used the variables **user** and **state**. When referencing a variable as another variable's value, we must add quotes around the value as shown in our example.

■ Conditions in Ansible

Whenever we have different scenarios, we put conditions according to the scenario.

■ When Statement

Sometimes you want to skip a particular command on a particular node.

```
--- # Condition Playbook
- hosts: demo <-- group name
   user: ansible
   become: yes <-- sudo privilege
   connection: ssh
   tasks:
            - name: Install Apache Server for Debian Family
               command: apt-get -y install apache2
               when: ansible_os_family == "Debian"
            - name: Install Apache Server for RedHat Family
               command: apt-get -y install apache2
               when: ansible_os_family == "RedHat"
```

■ Vault in Ansible

Ansible allows keeping sensitive data such as passwords or keys in encrypted files, rather than a plain-text in your playbooks. Ansible uses AES-256 Encryption Algorithm to encrypt the playbooks.

■ Creating a new encrypted playbook

→ `[ansible@ip]$ ansible-vault create <playbook_name>`

■ Edit the encrypted playbook

→ `[ansible@ip]$ ansible-vault edit <playbook_name>`

■ To change the password

→ `[ansible@ip]$ ansible-vault rekey <playbook_name>`

■ To encrypt an existing playbook

→ `[ansible@ip]$ ansible-vault encrypt <playbook_name>`

■ To decrypt an encrypted playbook

→ `[ansible@ip]$ ansible-vault decrypt <playbook_name>`

■ To run vault playbooks

→ `[ansible@ip]$ ansible-playbook --ask-vault-pass <playbook_name>`

█ Roles in Ansible

■ **Default** → It stores the data about the role/application. Default variable eg → If you want to run to port 80 or 8080 then variables need to define in this path.

■ Files → It contains files that need to be transferred to the remote VM (Static Files)

■ Handlers → They are triggers or tasks we can segregate all the handlers required in the playbook.

■ Meta → This directory contains files that establish roles dependencies, eg Author, Name, Supported Platform, Dependencies if any.

■ Tasks → It contains all the tasks that are normally in the playbook. eg → Installing packages and copying files etc.

■ Vars → Variables for the role can be specified in this directory and used in your configuration files for both vars and default stores variables.

█ Lab for Roles

Creating a structure for Roles and then using playbooks

```
[ansible@ip]$ ansible cd playbook/
[ansible@ip]$ touch roles/webserver/tasks/main.yml
[ansible@ip]$ touch master.yml
[ansible@ip]$ vi roles/webserver/tasks/main.yml [Inside main.yml]- name: install apache
  yum: pkg=httpd state=latest

[ansible@ip]$ vi master.yml--- # Master playbook for server
- hosts: demo <-- group name
   user: ansible
   become: yes <-- sudo privilege
   connection: ssh
   roles:
           - webserver[ansible@ip]$ ansible-playbook master.yml
```

█ Important points for Roles

We can use two techniques for reusing a set of tasks, ***Include,*** and ***Roles***

Roles are good for organizing tasks and encapsulating data needed to accomplish those tasks.

We can organize playbooks into a directory structure called roles.

Adding more and more functionality to the playbooks will make it difficult to maintain in a single file.

Ansible Best Practices to Follow

⚫ Generic & Project Structure Best Practices

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

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.

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.

⚫ 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 (opt ional)
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 for t he monitoring role
```

⚫ 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
```

⚫ Plays & Playbooks Best Practices

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_key =yes
   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.

```
- 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
```

⚫ 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
```

⚫ Modules Best Practices

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.

```yaml
- 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.

⚫ 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.