

Deep Dive Playbooks

Ansible Training

Agenda

- Lookups
- Conditions
- Loops
- Loops with Conditions
- Sequential vs Parallel Execution
- Multiplay files
- Tags
- Import

Agenda

- Ansible Facts
- Custom facts
- Conditionals
- Loops

Ansible Facts

Deep Dive Playbooks

Ansible Facts

- Facts in Ansible are essentially global variables that contain information about the system, like network interfaces or operating system
- They are derived from the host's system and environment
- Facts can be used in playbooks to make decisions, tailor configurations, or generate reports

Facts Gathering

- When you run a playbook, Ansible starts by gathering facts from the target system
- Once gathered, facts are stored in memory and can be referenced throughout the playbook
- Users can also define custom facts using local scripts on the target machine

Built-in Facts

- **ansible_architecture**: The architecture of the system (e.g., x86_64).
- **ansible_default_ipv4**: Information about the default IPv4 network interface.
- **ansible_distribution**: The name of the distribution (e.g., Ubuntu, CentOS).
- **ansible_distribution_version**: The version of the distribution (e.g., 18.04, 7).
- **ansible_hostname**: The hostname of the system.
- **ansible_interfaces**: A list of network interfaces on the system.
- **ansible_processor_vcpus**: The number of virtual CPUs.

Built-in Facts

- **ansible_virtualization_type**: The type of virtualization (e.g., kvm, qemu, vmware).
- **ansible_uptime**: System uptime.
- **ansible_users**: A list of user accounts on the system.
- **ansible_env**: Environment variables of the user executing Ansible.
- **ansible_mounts**: Information about mounted filesystems.
- **ansible_lsb**: Output of the lsb_release command if available.

How to Use Facts

- You can use facts on a similar way as you use variables
- All facts are stores on an array named **ansible_facts**

```
- name: Set hostname in config file
  lineinfile:
    path: /etc/myconfig.conf
    line: "hostname={{ ansible_facts['hostname'] }}"
```

Turn Off Fact Gathering

- If your playbook don't need any fact you can turn it off
- In your playbook, you can set the **gather_facts** parameter to **no**

```
- hosts: all
  gather_facts: no
  tasks:
    ...
```

Why Turn Off Fact Gathering

- **Performance:** Gathering facts can add overhead, especially when managing a large number of hosts. If facts aren't needed, disabling can speed up playbook execution
- **Irrelevance:** If your tasks don't depend on system-specific information, you can skip fact gathering
- **Custom Facts Only:** If you're only interested in a subset of facts or custom facts, you can gather those specifically and skip the default ones.
- **Avoiding Errors:** In some rare cases, the setup module might encounter errors on certain systems or configurations. Disabling fact gathering can be a temporary workaround.

Custom Facts

Deep Dive Playbooks

Custom Facts

- Custom facts in Ansible allow you to define and gather your own set of information from target systems, beyond the default facts that Ansible provides
- This can be particularly useful when you need to capture specific details about a system that aren't covered by the built-in facts

Custom Facts

- Custom facts are typically defined using scripts or static files placed in a specific directory on the target system.
- The default directory for custom facts is **/etc/ansible/facts.d**.
- Scripts can be written in any language but must be executable and return JSON output.
- Static **.fact** files can also be used, where each line is a key-value pair.

Gathering Custom Facts

- Using the **setup** Module
 - When you run the **setup** module (either implicitly at the start of a playbook or explicitly as a task), Ansible will automatically check the default directory (**/etc/ansible/facts.d**) for any custom fact scripts or files
- Once gathered, custom facts can be accessed in the same way as built-in facts, using the **ansible_facts** variable.
 - For example: **ansible_facts['ansible_local']['custom_fact_1']**
- Specifying a Different Directory
 - If you store custom facts in a directory other than the default, you can specify the directory using the **fact_path** parameter with the setup module:

```
- name: Gather facts from a custom directory
  setup:
    fact_path: "/path/to/custom/facts/directory"
```

Benefits Custom Facts

- **Flexibility:** Capture specific details tailored to your environment or application
- **Automation:** Use custom facts to drive conditional logic in your playbooks
- **Reporting:** Gather and report on specific metrics or configurations across your infrastructure

Considerations

- **Performance:** Ensure that custom fact scripts are efficient, especially if you have a large number of hosts
- **Security:** Be cautious about the information you capture and expose as custom facts, especially if sensitive

Gather Custom Facts

Demo

Conditionals

Deep Dive Playbooks

Conditionals

- Conditionals in Ansible allow you to control the execution of tasks based on specific criteria
- They are essential for creating dynamic and adaptive playbooks
- **when** Clause: The primary mechanism for conditionals in Ansible is the **when** keyword
- It determines if a task should be executed or skipped based on the evaluation of an expression
- You can use logical operators like **and**, **or**, and **not** to combine multiple conditions
- You can conditionally execute tasks based on the group membership of the target host
- Ansible facts, gathered using the setup module, can be used in conditionals to make decisions based on system characteristics.

Basic Conditional

- Execute a task only if a variable is defined

```
- name: Install nginx
  apt:
    name: nginx
    state: present
  when: nginx_install is defined and nginx_install
```

Multiple Conditions

- Execute a task based on multiple criteria using **and**.

```
- name: Install a package
  apt:
    name: "{{ package_name }}"
    state: present
  when:
    - package_name is defined
    - ansible_facts['os_family'] == 'Debian'
```

Group-based Conditional

- Execute a task only if the target host is a member of a specific group.

```
- name: Special configuration for database servers
  template:
    src: db_config.j2
    dest: /etc/db_config.conf
  when: "'dbservers' in group_names"
```

Fact-based Conditional

- Execute a task based on a system fact.

```
- name: Install package on Ubuntu only
  apt:
    name: some_package
    state: present
  when: ansible_facts['distribution'] == 'Ubuntu'
```


Task Results used in conditionals

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

```
- 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
```

Conditional with Failed Task

- Retry a task if it fails

```
- name: Attempt to fetch a file
  command: curl -O http://example.com/somefile.zip
  register: result
  ignore_errors: true

- name: Notify if fetch failed
  debug:
    msg: "File fetch failed!"
  when: result is failed
```

Failed_when


```
- name: Custom Failure Conditions with failed_when
hosts: localhost
tasks:
  - name: Execute the custom script
    command: /path/to/custom_script.sh
    register: script_result
    failed_when: script_result.rc == 2
    changed_when: script_result.rc != 0

  - name: Display script output
    debug:
      msg: "Script executed with exit code {{ script_result.rc }}"
```

Additional tips

- Using **| bool**:
 - When evaluating a condition that's expected to be a boolean, it's a good practice to use the bool filter, e.g., when: some_var | bool.
- **changed** and **failed**
 - After a task is executed, Ansible provides changed and failed attributes that can be used in subsequent conditionals.
- **failed_when**
 - This allows you to define custom failure conditions for a task based on its output.

Playbook with Conditional Tasks



```
- 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"
```

Loops

Deep Dive Playbooks

Loops

- Loops in Ansible allow you to iterate over a set of items and execute tasks multiple times based on those items
- They are essential for reducing code repetition and making playbooks more dynamic.
- Types of Loops in Ansible
 - Simple Loops: The basic loop structure, often used with lists.
 - Looping over Subelements: Useful for nested data structures.
 - Looping over Dictionary: Iterating over key-value pairs.
 - Looping over Files: Iterating over files in a directory.
 - Looping over Command Output: Using the result of a command as loop items.
 - Looping with Index: Accessing the current item's index during the loop

Simple Loop

- Install multiple packages using a list.

```
- hosts: localhost
  tasks:
    - name: Install multiple packages
      apt:
        name: "{{ item }}"
        state: present
      loop:
        - nginx
        - git
        - curl
```


Iterating over a list of hashes

- If you have a list of hashes, you can reference subkeys in a loop

```
- name: Add several users
  ansible.builtin.user:
    name: "{{ item.name }}"
    state: present
    groups: "{{ item.groups }}"
  loop:
    - { name: 'testuser1', groups: 'wheel' }
    - { name: 'testuser2', groups: 'root' }
```

Iterate over task result

```
- name: Iterate over .txt files in /tmp
hosts: localhost
tasks:
  - name: Find all .txt files in /tmp
    find:
      paths: "/tmp"
      patterns: "*.txt"
      recurse: no
      register: txt_files

  - name: Copy .txt files to /tmp/backup
    copy:
      src: "{{ item.path }}"
      dest: "/tmp/backup/{{ item.path | basename }}"
      loop: "{{ txt_files.files }}"
      when: txt_files.matched > 0
```

Looping over Dictionary

- Add multiple users with specific attributes

```
- hosts: localhost
  tasks:
    - name: Add users
      user:
        name: "{{ item.key }}"
        uid: "{{ item.value.uid }}"
        group: "{{ item.value.group }}"
        loop: "{{ lookup('dict', users) }}"
  vars:
    users:
      alice:
        uid: 1001
        group: admin
      bob:
        uid: 1002
        group: users
```

Looping Using Facts

- Conditional Execution: If you want to apply certain tasks only for a specific OS

```
- name: Install package X
  apt:
    name: package-X
    state: present
  when: ansible_facts['os_family'] == 'Debian'
```

- Dynamic Configuration: Using facts to tailor configurations:

```
- name: Set hostname in config file
  lineinfile:
    path: /etc/myconfig.conf
    line: "hostname={{ ansible_facts['hostname'] }}"
```

Control your loop

- You can use **loop_control** property to manage some behavior of your loop
- **loop_var**
 - Allows you to set a different name for the loop variable instead of the default item
- **index_var**
 - Provides the current iteration's index
- **pause**
 - Adds a delay (in seconds) between loop iterations.
- **end** and **start** (used together):
 - Control the start and end index for slicing the loop.

Loop control example

- To keep track of where you are in a loop, use the **index_var** directive with **loop_control**
- This directive specifies a variable name to contain the current loop index.

```

- name: Count our servers
  debug:
    msg: "{{ item }}" with index "{{ my_idx }}"
  loop:
    - server01
    - server02
    - server03
  loop_control:
    index_var: my_idx
```

Loops with Conditionals

Deep Dive Playbooks

Loops with conditionals

- Combining loops with conditionals in Ansible allows for powerful and dynamic playbook execution
- You can iterate over a set of items and conditionally decide whether to execute a task for each item based on specific criteria

Loops with **when** condition

- Define a when property within your loop to define a condition where an item should be considered or not
- This instruction allow you to control the items to be considered on the loop
- Makes your loop and playbook dynamic and based on any dynamic condition
- You may use any property that you may use on a simple conditional

Loops with **when** condition

```
packages:
  - name: nginx
    skip: no
  - name: apache2
    skip: yes
  - name: git
    skip: no

tasks:
  - name: Install packages
    apt:
      name: "[{ item.name }]"
      state: present
    loop: "[{ packages }]"
    when: item.skip == no
```

until loop control

- The until keyword in Ansible allows you to retry a task until a certain condition is met
- It's often used in conjunction with the **retries** and **delay** parameters to define how many times a task should be retried and the delay between retries

Wait for service to start

```
tasks:
  - name: Start the service
    service:
      name: myservice
      state: started

  - name: Wait for service to be running
    command: systemctl is-active myservice
    register: result
    retries: 5
    delay: 10
    until: result.stdout == 'active'
```

Wait for a URL to Respond

```
tasks:  
  - name: Check if the website is responding  
    uri:  
      url: http://example.com  
      status_code: 200  
    register: website_check  
    until: website_check.status == 200  
    retries: 10  
    delay: 5
```

Wait for a File to Exist

```
tasks:  
  - name: Wait for the file to be created  
    stat:  
      path: /path/to/file.txt  
    register: file_check  
    until: file_check.stat.exists  
    retries: 5  
    delay: 10
```

Demo: Loops & Conditions

Deep Dive Playbooks

Multiplays and Parallel Execution

Deep Dive Playbooks

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: webserver  
  serial: 2  
  tasks:  
    - name: Take out of load balancer pool  
      ...  
    - name: Update application  
      ...  
    - name: Add back to load balancer pool  
      ...
```

Demo: Multiplays

Deep Dive Playbooks

Handlers

Deep Dive Playbooks

Handlers

- Handlers in Ansible are special tasks used to perform actions like service restarts or configuration file updates in response to changes in tasks
- To trigger a handler, you use the **notify** keyword within tasks.
- When a task makes a change that needs to trigger a handler, it registers the handler to be executed.
- Handlers are only executed at the end of a play, after all tasks have run
- This helps prevent unnecessary restarts or changes during a play
- An handlers is queued if task when **notify** keyword is set returns a **changed** state

Handlers

- Handlers are idempotent, meaning they only execute once even if multiple tasks notify the same handler during a play
- This ensures that services are not restarted multiple times unnecessarily
- Handlers are executed in the order they are defined in the playbook, following the same playbook structure
- The order matters if you have dependencies between handlers.
- To force the execution of a handler, you can use the **meta** module with the **flush_handlers** argument

Handlers

- Handlers can have conditions using the **when** keyword
- This allows you to control when a handler is executed based on specific conditions, such as changes to a particular file or system state.
- Handlers are typically used to react to changes made during a playbook run
- For instance, if a configuration file is modified, a handler can be notified to restart the associated service

Handlers

```
- name: Configure and restart Nginx
  hosts: webserver_hosts
  become: yes
  tasks:
    - name: Copy Nginx configuration file
      copy:
        src: files/nginx.conf
        dest: /etc/nginx/nginx.conf
        notify: Restart Nginx
  handlers:
    - name: Restart Nginx
      service:
        name: nginx
        state: restarted
```

Demo: Handlers

Deep Dive Playbooks

Lookups Plugins

Deep Dive Playbooks

Lookup

- Lookups in Ansible are a way to query data from outside sources, allowing you to access and use this data within your playbooks
- Lookup plugins facilitate these queries
- Lookups allow you to pull in data from external sources into your Ansible playbooks
- This can be from a file, a database, a key-value store, an environment variable, and more.
- Usual format:

```
{{ lookup('plugin_name', 'parameter') }}
```

Lookup Plugins

- Lookup plugins are the backend mechanisms that power the lookup functionality.
- Ansible comes with a variety of built-in lookup plugins, and you can also create custom ones if needed
- You may get a list of available plugins

```
ansible-doc -t lookup --list
```

- And you may get more detail about each plugin

```
ansible-doc -t lookup file
```

Common Lookup Plugins

- file: Reads the contents of a file.

```
content: "{{ lookup('file', '/path/to/file.txt') }}"
```

- env: Reads the value of an environment variable

```
path: "{{ lookup('env', 'HOME') }}"
```

- password: Generates random passwords, often used for creating user accounts.

```
user_password: "{{ lookup('password', '/dev/null length=15 chars=ascii_letters') }}"
```


Common Lookup Plugins

- pipe: Executes a command and returns its output.

```
system_uptime: "{{ lookup('pipe', 'uptime') }}"
```

- ini: Fetches a specific key's value from an INI file.

```
db_user: "{{ lookup('ini', 'user section=database file=/path/to/config.ini') }}"
```

- redis_kv: Fetches values from a Redis database.

```
cache_value: "{{ lookup('redis_kv', 'my_key') }}"
```

Enhance Loops

- Lookups can be combined with loops to iterate over returned data
- For instance, using the **fileglob** lookup to loop over files:

```
- name: Copy all .conf files
  copy:
    src: "{{ item }}"
    dest: "/etc/config/"
  loop: "{{ lookup('fileglob', '/path/to/files/*.conf') }}"
```

Demo: Lookups

Deep Dive Playbooks

Tags

Deep Dive Playbooks

Tags

- Tags in Ansible playbooks are a powerful feature that provides granularity in playbook execution
- They allow you to run specific parts of a playbook without executing the entire set of tasks
- Selective Execution
 - Tags allow you to run specific tasks within a playbook, rather than executing everything
 - This is especially useful in large playbooks where you only want to run a subset of tasks
 - For example, if you have a playbook that configures a web server, database server, and firewall, you could use tags to only apply the firewall configurations without touching the web or database configurations

Why Using Tags

- Efficiency
 - Running a full playbook can be time-consuming
 - By using tags, you can significantly reduce the execution time by only running the tasks that are necessary for a particular change or update
 - This is especially beneficial in production environments where minimizing changes and disruptions is crucial
- Organization
 - Tags can serve as documentation, indicating the purpose or category of each task or set of tasks
 - They help in organizing playbooks and roles, making them more readable and maintainable

Why Using Tags

- Staging & Phased Rollouts
 - In complex deployments, you might want to stage or phase the rollout of changes
 - Tags allow you to break down the deployment process into stages, executing each stage separately.
 - For instance, in a software deployment, you might have tags like pre-deploy, deploy, and post-deploy to manage the deployment lifecycle.
- Flexibility
 - Tags offer flexibility in both development and operations. During development, you can use tags to test specific tasks without running the entire playbook.
 - In operations, tags allow ops teams to adapt to different scenarios, such as only running monitoring-related tasks or backup-related tasks as needed.

Usage

- Define tags on your playbook

```
tasks:
  - name: Install nginx
    apt:
      name: nginx
      state: present
    tags:
      - webserver
      - install
```

- Then use your command to execute or skip tags

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

```
ansible-playbook myplaybook.yml --skip-tags "install"
```


Demo: Tags

Deep Dive Playbooks

Import

Deep Dive Playbooks

Import

- In Ansible, the ability to import playbooks, tasks, or variables is crucial for creating modular, reusable, and maintainable automation code
- The import functionality allows users to break down complex automation tasks into smaller, more manageable pieces

Why use Import feature

- Modularity
 - By using import, you can break down large playbooks into smaller, more focused pieces
 - This modular approach makes it easier to understand, maintain, and update the automation code
 - For instance, instead of having a single monolithic playbook for setting up an entire application stack, you can have separate playbooks for the database, web server, and application logic, and then import them as needed
- Reusability
 - Once you've created a playbook, or task list, you can reuse it in multiple scenarios or projects by importing it
 - This reduces redundancy and ensures consistency across your automation tasks
 - For example, if you have a common set of tasks for basic server hardening, you can import those tasks into any playbook that sets up servers
- Maintainability
 - Smaller, modular pieces of code are easier to maintain
 - If a change is required, you can update the specific imported file without touching the main playbook or other unrelated tasks
 - This also makes it easier to track changes, troubleshoot issues, and understand the impact of modifications

Why use Import feature

- Collaboration
 - In team environments, different team members can work on separate pieces of the automation process
 - These pieces can then be imported into a main playbook, facilitating collaboration and parallel development
- Dynamic Inclusion
 - While import provides static inclusion (the imported tasks, playbooks, or roles are determined at playbook parsing time), it can be combined with variables to create dynamic paths, allowing for more flexible automation structures
- Organized Codebase
 - Using import helps in organizing the Ansible codebase
 - You can have a directory structure where related tasks, roles, or playbooks are grouped together, making it easier to navigate and manage the code
- Conditional Execution
 - You can combine import with conditionals (when clause) to determine whether to include certain tasks or playbooks based on specific conditions
 - This allows for adaptive playbook execution

Example: Import Tasks

- Directory structure

```
├─ main.yml
└─ tasks
    ├─ install_packages.yml
    └─ setup_firewall.yml
```

- main.yml

```
- hosts: localhost
  tasks:
    - name: Import tasks to install packages
      import_tasks: tasks/install_packages.yml

    - name: Import tasks to set up the firewall
      import_tasks: tasks/setup_firewall.yml
```

Example: Import Tasks

- install_packages.yml

```
- name: Install nginx
  apt:
    name: nginx
    state: present

- name: Install git
  apt:
    name: git
    state: present
```

- setup_firewall.yml

```
- name: Install UFW
  apt:
    name: ufw
    state: present

- name: Allow SSH through UFW
  ufw:
    rule: allow
    port: 22
    proto: tcp
```

Example: Import Playbooks

- Directory structure

```
├─ orchestrate.yml  
├─ setup_database.yml  
└─ setup_webserver.yml
```

- orchestrate.yml

```
- import_playbook: setup_database.yml  
- import_playbook: setup_webserver.yml
```


Demo: Import

Demo

