

Task Control

Loops

- In computer programming, a **loop** is a sequence of instructions that is continually repeated until a certain condition is reached.
- The purpose of **loops** is to repeat the same, or similar, code a number of times.
- Using loops saves administrators from the need to write multiple tasks that use the same module
- Ansible offers two keywords for creating loops: *loop* and *with_lookup*.
- Common Ansible loops include changing ownership on several files and/or directories with the `file` module, creating multiple users with the `user` module.

Simple Loops

- A simple loop iterates a task over a list of items.
- The `loop` keyword is added to the task, and takes as a value the list of items over which the task should be iterated.
- The loop variable `item` holds the value used during each iteration

```
---  
- name: Creating list of groups  
  become: true  
  gather_facts  
  hosts: localhost  
  tasks:  
    - name:  
      group:  
        name: "{{ item }}"  
        state: present  
      loop:  
        - grp1  
        - grp2  
        - grp3
```

Loops over a List of Hashes or Dictionaries

- The **loop** list does not need to be a list of simple values.
- In the following example, each item in the list is actually a hash or a dictionary.

Task Control

- Each hash or dictionary in the example has two keys, **name** and **uid**, and the value of each key in the current **item** loop variable can be retrieved with the `item.name` and `item.uid` variables, respectively.



```
---
- name: Create list of group
  hosts: localhost
  become: yes
  gather_facts: no
  tasks:
    - name: Creating group
      user:
        name: "{{item.name}}"
        uid: "{{item.uid}}"
        state: present
      loop:
        - name: user1
          uid: 2000
        - name: user2
          uid: 2001
        - name: user3
          uid: 2002
```

Using Register Variables with Loops

- To get the output with the help of register variables:

```
---
- name: Create list of group
  hosts: localhost
  become: yes
  gather_facts: no
  tasks:
    - name: Creating group
      user:
        name: "{{item.name}}"
        uid: "{{item.uid}}"
        state: present
      loop:
        - name: user1
          uid: 2000
        - name: user2
          uid: 2001
        - name: user3
          uid: 2002
      register: task_output
    - debug: var=task_output
```

Task Control

RUNNING TASKS CONDITIONALLY

- Ansible can use *conditionals* to execute tasks or plays when certain conditions are met.
- For example, a conditional can be used to determine available memory on a managed host before Ansible installs or configures a service.
- Conditionals allow administrators to differentiate between managed hosts and assign them functional roles based on the conditions that they meet.
- Playbook variables, registered variables, and Ansible facts can all be tested with conditionals.
- Operators to compare strings, numeric data, and Boolean values are available.

The following scenarios illustrate the use of conditionals in Ansible:

- A hard limit can be defined in a variable (for example, `min_memory`) and compared against the available memory on a managed host.
- The output of a command can be captured and evaluated by Ansible to determine whether or not a task completed before taking further action. For example, if a program fails, then a batch is skipped.
- Use Ansible facts to determine the managed host network configuration and decide which template file to send (for example, network bonding or trunking).
- The number of CPUs can be evaluated to determine how to properly tune a web server.
- Compare a registered variable with a predefined variable to determine if a service changed. For example, test the MD5 checksum of a service configuration file to see if the service is changed.

Conditional Task Syntax

- The **when** statement is used to run a task conditionally. It takes as a value the condition to test. If the condition is met, the task runs. If the condition is not met, the task is skipped.
- One of the simplest conditions that can be tested is whether a Boolean variable is true or false.

Task Control

- The **when** statement in the following example causes the task to run only if **run_my_task** is true:

Operation	Examples
Equal (String Value)	Ansible_machine == "x86_64"
Equal (Numeric Value)	max_memory == 512
Less Than	min_memory < 512
Greater Than	min_memory > 512
Less Than Equal To	min_memory <= 512
Greater Than Equal To	min_memory >= 512
Not equal to	min_memory != 512
Variable Exist	min_memory is defined
Variable does not Exist	min_memory is not defined
Boolean Variable is True	memory_available
Boolean variable is false	memory_available
First variable value is present as a value in second variable list	ansible_distribution in support_distros.

- Let understand the operators with the help of playbook:

```
---
- name: Understanding Operator
  gather_facts: false
  hosts: localhost
  vars:
    x: 10
    y: 20
  tasks:
    - debug:
        msg:
          - "The value of x: {{x}} and value of y: {{y}}"
          - "x==y: {{x==y}}"
          - "x!=y: {{x!=y}}"
          - "x>y: {{x>y}}"
          - "x>=y: {{x>=y}}"
          - "x<=y: {{x<=y}}"
          - "x<y: {{x<y}}"
```

Task Control

- In the example, the `ansible_distribution` variable is a fact determined during the **Gathering Facts** task, and identifies the managed host's operating system distribution.

```
---
- name: Demonstrate the "in" keyword
  hosts: localhost
  gather_facts: yes
  become: yes
  vars:
    distribution_list:
      - RedHat
      - Fedora
  tasks:
    - name: Install httpd using yum, where supported
      yum:
        name: httpd
        state: present
        when: ansible_distribution in distribution_list
```

- Example of Condition on basis of RAM, installing HTTPD package.

```
---
- name: Installing package with condition
  become: yes
  hosts: localhost
  vars:
  tasks:
    - name:
      debug:
        msg:
          "The value of ram is {{ansible_memtotal_mb}}"
    - name: Installing HTTPD
      yum:
        name: httpd
        state: present
        when: ansible_memtotal_mb >= 1024
```

Task Control

- Installing package on basis of OS family.

```
---  
- name: Understanding condition  
  hosts: public  
  become: yes  
  tasks:  
    - name: Install Httpd for Redhat  
      yum:  
        name: httpd  
        state: latest  
        when: ansible_os_family == "RedHat"  
    - name: Install apache for Debian  
      apt:  
        name: apache2  
        state: present  
        when: ansible_os_family == "Debian"
```

Testing Multiple Conditions

- One **when** statement can be used to evaluate multiple conditionals. To do so, conditionals can be combined with either the **and** or **or** keywords, and grouped with parentheses.
- If a conditional statement should be met when either condition is true, then you should use the **or** statement.
 - For example, the following condition is met if the machine is running either Red Hat Enterprise Linux or Fedora:

```
when: ansible_distribution == "RedHat" or ansible_distribution == "Fedora"
```

- With the **and** operation, both conditions have to be true for the entire conditional statement to be met.
 - For example, the following condition is met if the remote host is a Red Hat Enterprise Linux host, and the memory is the greater then 1GB

Task Control

```
when: ansible_memtotal_mb >= 1024 and ansible_distribution == "RedHat"
```

- The **when** keyword also supports using a list to describe a list of conditions. When a list is provided to the **when** keyword, all of the conditionals are combined using the **and** operation. The example below demonstrates another way to combine multiple conditional statements using the **and** operator:

When:

- ansible_distribution_version == "8.1"
- ansible_kernel == "4.18.0-147.el8.x86_64"

- More complex conditional statements can be expressed by grouping conditions with parentheses. This ensures that they are correctly interpreted.

When:>

```
( ansible_distribution == "RedHat" and
  ansible_distribution_major_version == "8" )
or
( ansible_distribution == "Fedora" and
  ansible_distribution_major_version == "30" )
```

```
---
- name: Installing package with condition
  become: yes
  hosts: localhost
  vars:
  tasks:
  - name:
    debug:
      msg:
        - "The value of ram is {{ansible_memtotal_mb}}"
        - "The Distribution is {{ansible_distribution}}"
  - name: Installing HTTPD
    yum:
      name: httpd
      state: present
#  when: ansible_memtotal_mb >= 1024 and ansible_distribution == "RedHat"
#  when: ansible_memtotal_mb >= 1024 or ansible_distribution == "Fedora"
```

Task Control

Running Conditional Task with Loop

- You can combine loops and conditionals.

```
---
- name:
  hosts: localhost
  become: yes
  gather_facts: yes
  tasks:
    - command: echo {{ item }}
      loop: [ 0, 2, 4, 6, 8, 10 ]
      when: item > 5
```

- In the following example, the *mariadb-server* package is installed by the **yum** module if there is a file system mounted on / with more than 300 MB free.
- The **ansible_mounts** fact is a list of dictionaries, each one representing facts about one mounted file system.
- The loop iterates over each dictionary in the list, and the conditional statement is not met unless a dictionary is found representing a mounted file system where both conditions are true.

```
---
- name: Understanding condition with loop concept
  hosts: localhost
  become: yes
  gather_facts: true
  tasks:
    - name: install mariadb-server if enough space on root
      yum:
        name: mariadb-server
        state: latest
      loop: "{{ ansible_mounts }}"
      when: item.mount == "/" and item.size_available > 300000000
      register: results
    - debug: var=results
```


Task Control

Ansible Handlers

- Handlers are just like regular tasks in an Ansible playbook, but are only run if the Task contains a “notify” directive and also indicates that it changed something.
- For example, if a configuration file is changed then the task referencing the configuration file templating operation may notify a service restart handler. This means services can be bounced only if they need to be restarted.
- Handlers can be used for things other than service restarts, but service restarts are the most common usage.
- Each handler has a globally unique name and is triggered at the end of a block of tasks in a playbook.
- If no task notifies the handler by name then the handler will not run.
- If one or more tasks notify the handler, the handler will run exactly once after all other tasks in the play have completed.

Some Important about Handlers

As discussed in the Ansible documentation, there are some important things to remember about using handlers:

- Handlers always run in the order specified by the **handlers** section of the play. They do not run in the order in which they are listed by **notify** statements in a task, or in the order in which tasks notify them.
- Handlers normally run after all other tasks in the play complete. A handler called by a task in the **tasks** part of the playbook will not run until *all* tasks under **tasks** have been processed. (There are some minor exceptions to this.)
- Handler names exist in a per-play namespace. If two handlers are incorrectly given the same name, only one will run.
- Even if more than one task notifies a handler, the handler only runs once. If no tasks notify it, a handler will not run.
- If a task that includes a **notify** statement does not report a **changed** result (for example, a package is already installed and the task reports **ok**), the handler is not notified. The handler is skipped unless another task notifies it. Ansible notifies handlers only if the task reports the **changed** status.

Task Control

```
---
- name: configure ssh file
  become: yes
  gather_facts: false
  hosts: localhost
  tasks:
    - name: replace content
      replace:
        path: /etc/ssh/sshd_config
        regexp: PasswordAuthentication yes
        replace: PasswordAuthentication no
      notify: "restart ssh server"
  handlers:
    - name: "Restart SSH"
      service:
        name: sshd
        state: restarted
      listen: "restart ssh server"
```

MANAGING TASK ERRORS IN PLAYS

- Ansible evaluates the return code of each task to determine whether the task succeeded or failed. Normally, when a task fails Ansible immediately aborts the rest of the play on that host, skipping all subsequent tasks.
- However, sometimes you might want to have play execution continue even if a task fails.
- There are a number of Ansible features that can be used to manage task errors.

Ignoring Task Failure

- By default, if a task fails, the play is aborted. However, this behaviour can be overridden by ignoring failed tasks. You can use the `ignore_errors` keyword in a task to accomplish this.
- Let's understand ignoring task with playbook.

Task Control

```
---
- name: Task Failure Exercise
  hosts: localhost
  gather_facts: no
  become: yes
  vars:
    web_package: http
    db_package: mariadb-server
    db_service: mariadb
  tasks:
    - name: Install {{ web_package }} package
      yum:
        name: "{{ web_package }}"
        state: present
        ignore_errors: yes

    - name: Install {{ db_package }} package
      yum:
        name: "{{ db_package }}"
        state: absent
```

Forcing Execution of Handlers after Task Failure

- Normally when a task fails and the play aborts on that host, any handlers that had been notified by earlier tasks in the play will not run. If you set the **force_handlers: yes** keyword on the play, then notified handlers are called even if the play aborted because a later task failed.

Specifying Task Failure Conditions

- You can use the **failed_when** keyword on a task to specify which conditions indicate that the task has failed. This is often used with command modules that may successfully execute a command, but the command's output indicates a failure.

Specifying When a Task Reports “Changed” Results

- When a task makes a change to a managed host, it reports the **changed** state and notifies handlers. When a task does not need to make a change, it reports **ok** and does not notify handlers.

Task Control

Ansible Blocks and Error Handling

- In playbooks, *blocks* are clauses that logically group tasks, and can be used to control how tasks are executed. For example, a task block can have a **when** keyword to apply a conditional to multiple tasks:
- Blocks also allow for error handling in combination with the **rescue** and **always** statements.
- **block**: Defines the main tasks to run.
- **rescue**: Defines the tasks to run if the tasks defined in the **block** clause fail.
- **always**: Defines the tasks that will always run independently of the success or failure of tasks defined in the **block** and **rescue** clauses.
- Let's understand how block and rescue work

```
---
- name: block and rescue
  hosts: localhost
  gather_facts: no
  become: yes
  vars:
    web_package: http
    db_package: mariadb-server
    db_service: mariadb
  tasks:
    - name: If block is success then rescue will not run
      block:
        - name: Install {{ web_package }} package
          yum:
            name: "{{ web_package }}"
            state: present
      rescue:
        - name: Install {{ db_package }} package
          yum:
            name: "{{ db_package }}"
            state: present
```

Task Control

- Let's write down a play book what we have learn about Ansible Handlers .

```
---
- name: "Working with Handlers play1"
  become: yes
  gather_facts: false
  hosts: localhost
  tasks:
    - name: working with ssh # task 1
      replace:
        path: /etc/ssh/sshd_config
        regexp: PasswordAuthentication yes
        replace: PasswordAuthentication no
      notify: "restart ssh server"
  handlers:
    - name: "Restart SSH" # task 2
      service:
        name: sshd
        state: restarted
      listen: "restart ssh server"
- name: "Working with error handling play2"
  become: yes
  hosts: localhost
  vars:
    web_package: httpd
    db_package: mariadb-server
    db_service: mariadb
  tasks:
    - name: working with block and rescue
      block:
        - name: check local time
          command: date
          register: results
          changed_when: false
        - name: print date
          debug:
            var: results.stdout
        - name: Install {{ web_package }} package
          yum:
            name: "{{ web_package }}"
            state: present
            failed_when: web_package == "httpd"
      rescue:
        - name: Install {{ db_package }} package
          yum:
            name: "{{ db_package }}"
            state: present
      always:
        - name: Start the service
          service:
            name: "{{ db_service }}"
            state: started
```