Conditionals

In a playbook, you may want to execute different tasks, or have different goals, depending on the value of a fact (data about the remote system), a variable, or the result of a previous task. You may want the value of some variables to depend on the value of other variables. Or you may want to create additional groups of hosts based on whether the hosts match other criteria. You can do all of these things with conditionals.

Ansible uses Jinja2 tests (playbooks_tests.html#playbooks-tests) and filters (playbooks_filters.html#playbooks-filters) in conditionals. Ansible supports all the standard tests and filters, and adds some unique ones as well.

Note

There are many options to control execution flow in Ansible. You can find more examples of supported conditionals at https://jinja.palletsprojects.com/en/latest/templates/#comparisons (https://jinja.palletsprojects.com/en/latest/templates/#comparisons).

Search this site

Basic conditionals with `when`

The simplest conditional statement applies to a single task. Create the task, then add a `when` statement that applies a test. The `when` clause is a raw Jinja2 expression without double curly braces (see group_by_module (https://docs.ansible.com/ansible/7/collections/ansible/builtin/group_by_module.html#group-by-module)). When you run the task or playbook, Ansible evaluates the test for all hosts. On any host where the test passes (returns a value of True), Ansible runs that task. For example, if you are installing mysql on multiple machines, some of which have SELinux enabled, you might have a task to configure SELinux to allow mysql to run. You would only want that task to run on machines that have SELinux enabled:

```
tasks:
  - name: Configure SELinux to start mysql on any port
    ansible.posix.seboolean:
      name: mysql_connect_any
      state: true
      persistent: true
    when: ansible_selinux.status == "enabled"
    # all variables can be used directly in conditionals without double curly braces
```

Conditionals based on ansible_facts

Often you want to execute or skip a task based on facts. Facts are attributes of individual hosts, including IP address, operating system, the status of a filesystem, and many more. With conditionals based on facts:

- You can install a certain package only when the operating system is a particular version.
- You can skip configuring a firewall on hosts with internal IP addresses.
- You can perform cleanup tasks only when a filesystem is getting full.

See Commonly-used facts for a list of facts that frequently appear in conditional statements. Not all facts exist for all hosts. For example, the 'lsb_major_release' fact used in an example below only exists when the lsb_release package is installed on the target host. To see what facts are available on your systems, add a debug task to your playbook:

```
- name: Show facts available on the system
  ansible.builtin.debug:
    var: ansible_facts
```

Here is a sample conditional based on a fact:

```
tasks:
  - name: Shut down Debian flavored systems
    ansible.builtin.command: /sbin/shutdown -t now
    when: ansible_facts['os_family'] == "Debian"
```

If you have multiple conditions, you can group them with parentheses:

```
tasks:
  - name: Shut down CentOS 6 and Debian 7 systems
    ansible.builtin.command: /sbin/shutdown -t now
    when: (ansible_facts['distribution'] == "CentOS" and
ansible_facts['distribution_major_version'] == "6") or
          (ansible_facts['distribution'] == "Debian" and
ansible_facts['distribution_major_version'] == "7")
```

You can use logical operators (https://jinja.palletsprojects.com/en/latest/templates/#logic) to combine conditions. When you have multiple conditions that all need to be true (that is, a logical `and`), you can specify them as a list:

```
tasks:
  - name: Shut down CentOS 6 systems
    ansible.builtin.command: /sbin/shutdown -t now
    when:
      - ansible_facts['distribution'] == "CentOS"
      - ansible_facts['distribution_major_version'] == "6"
```

If a fact or variable is a string, and you need to run a mathematical comparison on it, use a filter to ensure that Ansible reads the value as an integer:

```
tasks:
  - ansible.builtin.shell: echo "only on Red Hat 6, derivatives, and later"
    when: ansible_facts['os_family'] == "RedHat" and ansible_facts['lsb']
['major_release'] | int >= 6
```

Conditions based on registered variables

Often in a playbook you want to execute or skip a task based on the outcome of an earlier task. For example, you might want to configure a service after it is upgraded by an earlier task. To create a conditional based on a registered variable:

1. Register the outcome of the earlier task as a variable.
2. Create a conditional test based on the registered variable.

You create the name of the registered variable using the `register` keyword. A registered variable always contains the status of the task that created it as well as any output that task generated. You can use registered variables in templates and action lines as well as in conditional `when` statements. You can access the string contents of the registered variable using `variable.stdout`. For example:

```
- name: Test play
  hosts: all

  tasks:

      - name: Register a variable
        ansible.builtin.shell: cat /etc/motd
        register: motd_contents

      - name: Use the variable in conditional statement
        ansible.builtin.shell: echo "motd contains the word hi"
        when: motd_contents.stdout.find('hi') != -1
```

You can use registered results in the loop of a task if the variable is a list. If the variable is not a list, you can convert it into a list, with either `stdout_lines` or with `variable.stdout.split()`. You can also split the lines by other fields:

```
- name: Registered variable usage as a loop list
  hosts: all
  tasks:

    - name: Retrieve the list of home directories
      ansible.builtin.command: ls /home
      register: home_dirs

    - name: Add home dirs to the backup spooler
      ansible.builtin.file:
        path: /mnt/bkspool/{{ item }}
        src: /home/{{ item }}
        state: link
      loop: "{{ home_dirs.stdout_lines }}"
      # same as loop: "{{ home_dirs.stdout.split() }}"
```

The string content of a registered variable can be empty. If you want to run another task only on hosts where the stdout of your registered variable is empty, check the registered variable's string contents for emptiness:

```
- name: check registered variable for emptiness
  hosts: all

  tasks:

      - name: List contents of directory
        ansible.builtin.command: ls mydir
        register: contents

      - name: Check contents for emptiness
        ansible.builtin.debug:
          msg: "Directory is empty"
        when: contents.stdout == ""
```

Ansible always registers something in a registered variable for every host, even on hosts where a task fails or Ansible skips a task because a condition is not met. To run a follow-up task on these hosts, query the registered variable for `is skipped` (not for "undefined" or "default"). See Registering variables (playbooks_variables.html#registered-variables) for more information. Here are sample conditionals based on the success or failure of a task. Remember to ignore errors if you want Ansible to continue executing on a host when a failure occurs:

Search this site

```
tasks:
  - name: Register a variable, ignore errors and continue
    ansible.builtin.command: /bin/false
    register: result
    ignore_errors: true

  - name: Run only if the task that registered the "result" variable fails
    ansible.builtin.command: /bin/something
    when: result is failed

  - name: Run only if the task that registered the "result" variable succeeds
    ansible.builtin.command: /bin/something_else
    when: result is succeeded

  - name: Run only if the task that registered the "result" variable is skipped
    ansible.builtin.command: /bin/still/something_else
    when: result is skipped

  - name: Run only if the task that registered the "result" variable changed something.
    ansible.builtin.command: /bin/still/something_else
    when: result is changed
```

Note

Older versions of Ansible used `success` and `fail`, but `succeeded` and `failed` use the correct tense. All of these options are now valid.

Conditionals based on variables

You can also create conditionals based on variables defined in the playbooks or inventory. Because conditionals require boolean input (a test must evaluate as True to trigger the condition), you must apply the `| bool` filter to non boolean variables, such as string variables with content like ‘yes’, ‘on’, ‘1’, or ‘true’. You can define variables like this:

```
vars:
  epic: true
  monumental: "yes"
```

With the variables above, Ansible would run one of these tasks and skip the other:

```
tasks:
    - name: Run the command if "epic" or "monumental" is true
      ansible.builtin.shell: echo "This certainly is epic!"
      when: epic or monumental | bool

    - name: Run the command if "epic" is false
      ansible.builtin.shell: echo "This certainly isn't epic!"
      when: not epic
```

Search this site

If a required variable has not been set, you can skip or fail using Jinja2's *defined* test. For example:

```
tasks:
    - name: Run the command if "foo" is defined
      ansible.builtin.shell: echo "I've got '{{ foo }}' and am not afraid to use it!"
      when: foo is defined

    - name: Fail if "bar" is undefined
      ansible.builtin.fail: msg="Bailing out. This play requires 'bar'"
      when: bar is undefined
```

This is especially useful in combination with the conditional import of vars files (see below). As the examples show, you do not need to use *{{ }}* to use variables inside conditionals, as these are already implied.

Using conditionals in loops

If you combine a `when` statement with a loop (playbooks_loops.html#playbooks-loops), Ansible processes the condition separately for each item. This is by design, so you can execute the task on some items in the loop and skip it on other items. For example:

```
tasks:
    - name: Run with items greater than 5
      ansible.builtin.command: echo {{ item }}
      loop: [ 0, 2, 4, 6, 8, 10 ]
      when: item > 5
```

If you need to skip the whole task when the loop variable is undefined, use the *|default* filter to provide an empty iterator. For example, when looping over a list:

```
- name: Skip the whole task when a loop variable is undefined
  ansible.builtin.command: echo {{ item }}
  loop: "{{ mylist|default([]) }}"
  when: item > 5
```

You can do the same thing when looping over a dict:

```
- name: The same as above using a dict
  ansible.builtin.command: echo {{ item.key }}
  loop: "{{ query('dict', mydict|default({})) }}"
  when: item.value > 5
```

Search this site

Loading custom facts

You can provide your own facts, as described in Should you develop a module? (../dev_guide/developing_modules.html#developing-modules). To run them, just make a call to your own custom fact gathering module at the top of your list of tasks, and variables returned there will be accessible to future tasks:

```
tasks:
    - name: Gather site specific fact data
      action: site_facts

    - name: Use a custom fact
      ansible.builtin.command: /usr/bin/thingy
      when: my_custom_fact_just_retrieved_from_the_remote_system == '1234'
```

Conditionals with re-use

You can use conditionals with re-usable tasks files, playbooks, or roles. Ansible executes these conditional statements differently for dynamic re-use (includes) and for static re-use (imports). See Re-using Ansible artifacts (playbooks_reuse.html#playbooks-reuse) for more information on re-use in Ansible.

Conditionals with imports

When you add a conditional to an import statement, Ansible applies the condition to all tasks within the imported file. This behavior is the equivalent of Tag inheritance: adding tags to multiple tasks (playbooks_tags.html#tag-inheritance). Ansible applies the condition to every task, and evaluates each task separately. For example, if you want to define and then display a variable that was not previously defined, you might have a playbook called `main.yml` and a tasks file called `other_tasks.yml`:

```
# all tasks within an imported file inherit the condition from the import statement
# main.yml
- hosts: all
  tasks:
  - import_tasks: other_tasks.yml # note "import"
    when: x is not defined

# other_tasks.yml
- name: Set a variable
  ansible.builtin.set_fact:
    x: foo

- name: Print a variable
  ansible.builtin.debug:
    var: x
```

Ansible expands this at execution time to the equivalent of:

Search this site

```
- name: Set a variable if not defined
  ansible.builtin.set_fact:
    x: foo
  when: x is not defined
  # this task sets a value for x

- name: Do the task if "x" is not defined
  ansible.builtin.debug:
    var: x
  when: x is not defined
  # Ansible skips this task, because x is now defined
```

If `x` is initially defined, both tasks are skipped as intended. But if `x` is initially undefined, the debug task will be skipped since the conditional is evaluated for every imported task. The conditional will evaluate to `true` for the `set_fact` task, which will define the variable and cause the `debug` conditional to evaluate to `false`.

If this is not the behavior you want, use an `include_*` statement to apply a condition only to that statement itself.

```
# using a conditional on include_* only applies to the include task itself
# main.yml
- hosts: all
  tasks:
  - include_tasks: other_tasks.yml # note "include"
    when: x is not defined
```

Now if `x` is initially undefined, the debug task will not be skipped because the conditional is evaluated at the time of the include and does not apply to the individual tasks.

You can apply conditions to `import_playbook` as well as to the other `import_*` statements. When you use this approach, Ansible returns a 'skipped' message for every task on every host that does not match the criteria, creating repetitive output. In many cases the group_by module (../collections/ansible/builtin/group_by_module.html#group-by-module) can be a more streamlined way to accomplish the same objective; see Handling OS and distro differences (../tips_tricks/ansible_tips_tricks.html#os-variance).

Conditionals with includes

When you use a conditional on an `include_*` statement, the condition is applied only to the include task itself and not to any other tasks within the included file(s). To contrast with the example used for conditionals on imports above, look at the same playbook and tasks file, but using an include instead of an import:

```
# Includes let you re-use a file to define a variable when it is not already defined

# main.yml
- include_tasks: other_tasks.yml
  when: x is not defined

# other_tasks.yml
- name: Set a variable
  ansible.builtin.set_fact:
    x: foo

- name: Print a variable
  ansible.builtin.debug:
    var: x
```

Ansible expands this at execution time to the equivalent of:

```
# main.yml
- include_tasks: other_tasks.yml
  when: x is not defined
  # if condition is met, Ansible includes other_tasks.yml

# other_tasks.yml
- name: Set a variable
  ansible.builtin.set_fact:
    x: foo
  # no condition applied to this task, Ansible sets the value of x to foo

- name: Print a variable
  ansible.builtin.debug:
    var: x
  # no condition applied to this task, Ansible prints the debug statement
```

By using `include_tasks` instead of `import_tasks`, both tasks from `other_tasks.yml` will be executed as expected. For more information on the differences between `include` v `import` see Re-using Ansible artifacts (playbooks_reuse.html#playbooks-reuse).

Conditionals with roles

There are three ways to apply conditions to roles:

- Add the same condition or conditions to all tasks in the role by placing your `when` statement under the `roles` keyword. See the example in this section.
- Add the same condition or conditions to all tasks in the role by placing your `when` statement on a static `import_role` in your playbook.
- Add a condition or conditions to individual tasks or blocks within the role itself. This is the only approach that allows you to select or skip some tasks within the role based on your `when` statement. To select or skip tasks within the role, you must have conditions set on individual tasks or blocks, use the dynamic `include_role` in your playbook, and add the condition or conditions to the include. When you use this approach, Ansible applies the condition to the include itself plus any tasks in the role that also have that `when` statement.

When you incorporate a role in your playbook statically with the `roles` keyword, Ansible adds the conditions you define to all the tasks in the role. For example:

```
- hosts: webservers
  roles:
     - role: debian_stock_config
       when: ansible_facts['os_family'] == 'Debian'
```

Selecting variables, files, or templates based on facts

Sometimes the facts about a host determine the values you want to use for certain variables or even the file or template you want to select for that host. For example, the names of packages are different on CentOS and on Debian. The configuration files for common services are also different on different OS flavors and versions. To load different variables file, templates, or other files based on a fact about the hosts:

1. name your vars files, templates, or files to match the Ansible fact that differentiates them
2. select the correct vars file, template, or file for each host with a variable based on that Ansible fact

Ansible separates variables from tasks, keeping your playbooks from turning into arbitrary code with nested conditionals. This approach results in more streamlined and auditable configuration rules because there are fewer decision points to track.

Selecting variables files based on facts

You can create a playbook that works on multiple platforms and OS versions with a minimum of syntax by placing your variable values in vars files and conditionally importing them. If you want to install Apache on some CentOS and some Debian servers, create variables files with YAML keys and values. For example:

```
---
# for vars/RedHat.yml
apache: httpd
somethingelse: 42
```

Then import those variables files based on the facts you gather on the hosts in your playbook:

```
---
- hosts: webservers
  remote_user: root
  vars_files:
    - "vars/common.yml"
    - [ "vars/{{ ansible_facts['os_family'] }}.yml", "vars/os_defaults.yml" ]
  tasks:
  - name: Make sure apache is started
    ansible.builtin.service:
      name: '{{ apache }}'
      state: started
```

Ansible gathers facts on the hosts in the webservers group, then interpolates the variable "ansible_facts['os_family']" into a list of filenames. If you have hosts with Red Hat operating systems (CentOS, for example), Ansible looks for 'vars/RedHat.yml'. If that file does not exist, Ansible attempts to load 'vars/os_defaults.yml'. For Debian hosts, Ansible first looks for 'vars/Debian.yml', before falling back on 'vars/os_defaults.yml'. If no files in the list are found, Ansible raises an error.

Selecting files and templates based on facts

You can use the same approach when different OS flavors or versions require different configuration files or templates. Select the appropriate file or template based on the variables assigned to each host. This approach is often much cleaner than putting a lot of conditionals into a single template to cover multiple OS or package versions.

For example, you can template out a configuration file that is very different between, say, CentOS and Debian:

```
- name: Template a file
  ansible.builtin.template:
    src: "{{ item }}"
    dest: /etc/myapp/foo.conf
  loop: "{{ query('first_found', { 'files': myfiles, 'paths': mypaths}) }}"
  vars:
    myfiles:
      - "{{ ansible_facts['distribution'] }}.conf"
      -  default.conf
    mypaths: ['search_location_one/somedir/', '/opt/other_location/somedir/']
```

Debugging conditionals

If your conditional `when` statement is not behaving as you intended, you can add a `debug` statement to determine if the condition evaluates to `true` or `false`. A common cause of unexpected behavior in conditionals is testing an integer as a string or a string as an integer. To debug a conditional statement, add the entire statement as the `var:` value in a `debug` task. Ansible then shows the test and how the statement evaluates. For example, here is a set of tasks and sample output:

```
- name: check value of return code
  ansible.builtin.debug:
    var: bar_status.rc

- name: check test for rc value as string
  ansible.builtin.debug:
    var: bar_status.rc == "127"

- name: check test for rc value as integer
  ansible.builtin.debug:
    var: bar_status.rc == 127
```

```
TASK [check value of return code] *********************************************************************************
ok: [foo-1] => {
    "bar_status.rc": "127"
}

TASK [check test for rc value as string] **************************************************************************
ok: [foo-1] => {
    "bar_status.rc == \"127\"": false
}

TASK [check test for rc value as integer] *************************************************************************
ok: [foo-1] => {
    "bar_status.rc == 127": true
}
```

Commonly-used facts

The following Ansible facts are frequently used in conditionals.

ansible_facts['distribution']

Possible values (sample, not complete list):

Search this site

```
Alpine
Altlinux
Amazon
Archlinux
ClearLinux
Coreos
CentOS
Debian
Fedora
Gentoo
Mandriva
NA
OpenWrt
OracleLinux
RedHat
Slackware
SLES
SMGL
SUSE
Ubuntu
VMwareESX
```

ansible_facts['distribution_major_version']

The major version of the operating system. For example, the value is *16* for Ubuntu 16.04.

ansible_facts['os_family']

Possible values (sample, not complete list):

```
AIX
Alpine
Altlinux
Archlinux
Darwin
Debian
FreeBSD
Gentoo
HP-UX
Mandrake
RedHat
SGML
Slackware
Solaris
Suse
Windows
```

See also

Working with playbooks (playbooks.html#working-with-playbooks)
An introduction to playbooks

Roles (playbooks_reuse_roles.html#playbooks-reuse-roles)
Playbook organization by roles

General tips (../tips_tricks/ansible_tips_tricks.html#tips-and-tricks)
: Tips and tricks for playbooks

Using Variables (playbooks_variables.html#playbooks-variables)
: All about variables

User Mailing List (https://groups.google.com/group/ansible-devel)
: Have a question? Stop by the google group!

Real-time chat (../community/communication.html#communication-irc)
: How to join Ansible chat channels