ANSIBLE

How to Use Different Types of Ansible Variables (Examples)

Ioannis Moustakis

Updated 11 May 2023 · 14 min read

Ansible variables are dynamic values used within Ansible playbooks and roles to enable customization, flexibility, and reusability of configurations. They are very similar to variables in programming languages, helping you manage complex tasks more efficiently by allowing the same playbook or role to be applied across different environments, systems, or contexts without the need for hardcoding specific information.

Ansible Tutorial or Working with Ansible Playbooks blog posts. You can find this article's code on this repository if you wish to follow along.

Why Variables Are Useful in Ansible

The use of variables simplifies the management of dynamic values throughout an Ansible project and can potentially reduce the number of human errors. We have a convenient way to handle variations and differences between different environments and systems with variables.

Another advantage of variables in Ansible is that we have the flexibility to define them in multiple places with different precedence according to our use case. We can also register new variables in our playbooks by using the returned value of a task.

Ansible facts are a special type of variables that Ansible retrieves from any remote host for us to leverage them in Ansible projects. For example, we can get information regarding the operating system distribution with *ansible_distribution*, information about devices on the host, the python version that Ansible is using with *ansible_python_version*, and the system architecture, among others. To access this data, we have to reference the *ansible_facts* variable.

Variable Name Rules

Ansible has a strict set of rules to create valid variable names. Variable names can contain only letters, numbers, and underscores and must start with a letter or underscore. Some

Defining and Referencing Simple Variables

The simplest use case of variables is to define a variable name with a single value using standard YAML syntax. Although this pattern can be used in many places, we will show an example in a playbook for simplicity.

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

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

In the above example, after the **vars** block, we define the variable **username**, and assign the value **bob**. Later, to reference the value in the task, we use Jinja2 syntax like this “{{ username }}”

If a variable’s value starts with curly braces, we must quote the whole expression to allow YAML to interpret the syntax correctly.

List, Dictionary & Nested Variables

```
vars:
  version:
    - v1
    - v2
    - v3
```

To reference a specific value from the list we must select the correct field. For example, to access the third value *v3*:

```
version: "{{ version[2] }}"
```

Another useful option is to store key-value pairs in variables as dictionaries. For example:

```
vars:
  users:
    - user_1: maria
    - user_2: peter
    - user_3: sophie
```

Similarly, to reference the third field from the dictionary, use the bracket or dot notation:

```
users['user_3']
```

Note that the bracket notation is preferred as you might encounter problems using the dot notation in special cases.

Sometimes, we have to create or use nested variable structures. For example, facts are nested data structures. We have to use a bracket or dot notation to reference nested variables.

```
vars:
  cidr_blocks:
      production:
        vpc_cidr: "172.31.0.0/16"
      staging:
        vpc_cidr: "10.0.0.0/24"

tasks:
- name: Print production vpc_cidr
  ansible.builtin.debug:
    var: cidr_blocks['production']['vpc_cidr']
```

Special Variables

Ansible special variables are a set of predefined variables that contain information about the system data, inventory, or execution context inside an Ansible playbook or role. These include magic variables, connection variables, and facts. The names of these variables are reserved.

Magic Variables

```
---
- name: Echo playbook
  hosts: localhost
  gather_facts: no
  tasks:
    - name: Echo inventory_hostname
      ansible.builtin.debug:
        msg:
          - "Hello from Ansible playbook!"
          - "This is running on {{ inventory_hostname }}"
```

In the above playbook, we are defining a playbook that uses the *inventory_hostname* magic variable. We are using this variable to get the name of the host on which Ansible runs and print a message with it as shown below.

```
PLAY [Echo playbook] ***********************************************************

TASK [Echo inventory_hostname] *************************************************
ok: [localhost] => {
    "msg": [
        "Hello from Ansible playbook!",
        "This is running on localhost"
    ]
}

PLAY RECAP *********************************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0    skipped
```

- **play_hosts** → lists all the hosts that are targeted by the current play.
- **group_names** → contains a list of groups names to which the current host belongs in the inventory.
- **groups** → key/value pair of all the groups in the inventory with all the hosts that belong to each group.

Ansible Facts

Ansible facts are leveraged for getting system and hardware facts gathered about the current host during playbook execution. This data is often utilized for creating dynamic inventories, templating, or making decisions based on host-specific attributes. The gathered facts can be accessed using the *ansible_facts* variable, allowing you to reference specific information like the operating system, IP address, or CPU architecture.

To collect facts about a specific host, you can run the following command:

```
ansible -m setup <hostname>
```

Also, you can add the `gather_facts: yes` option to your playbook to ensure facts are collected after executing tasks.

With Ansible facts, you have the option to filter for specific facts like os or even the IP address. This can be done with:

```
ansible -m setup <hostname> -a "filter=<fact_name>
```

flexibility in managing various connection types, authentication methods, and host-specific configurations.

```
---
- name: Echo message on localhost
  hosts: localhost
  connection: local
  gather_facts: no
  vars:
    message: "Hello from Ansible playbook on localhost!"
  tasks:
    - name: Echo message and connection type
      ansible.builtin.shell: "echo '{{ message }}' ; echo 'Connection type: {{ ansible
      register: echo_output

    - name: Display output
      ansible.builtin.debug:
        msg: "{{ echo_output.stdout_lines }}"
```

In the above example, we are using the connection variable, to show the connection type on a run as shown below:

```
PLAY [Echo message on local host] ***********************************************

TASK [Echo message and connection type] *****************************************
changed: [localhost]

TASK [Display output] ***********************************************************
ok: [localhost] => {
    "msg": [
```

```
PLAY RECAP *********************************************************************
localhost                  : ok=1    changed=0    unreachable=0    failed=0    skipped
```

Registering Variables

During our plays, we might find it handy to utilize the output of a task as a variable that we can use in the following tasks. We can use the keyword **register** to create our own custom variables from task output.

```
- name: Example Register Variable Playbook
  hosts: all

  tasks:
  - name: Run a script and register the output as a variable
    shell: "find hosts"
    args:
      chdir: "/etc"
    register: find_hosts_output
  - name: Use the output variable of the previous task
    debug:
      var: find_hosts_output
```

In the above example, we register the output of the command *find /etc/hosts*, and we showcase how we can use the variable in the next task by printing its value.

```
TASK [Run a script and register the output as a variable] ***********************************************
changed: [host1]
changed: [host2]

TASK [Use the output variable of the previous task] *****************************************************
ok: [host1] => {
    "find_hosts_output": {
        "changed": true,
        "cmd": "find hosts",
        "delta": "0:00:00.002781",
        "end": "2022-04-01 19:34:59.734654",
        "failed": false,
        "rc": 0,
        "start": "2022-04-01 19:34:59.731873",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "hosts",
        "stdout_lines": [
            "hosts"
        ]
    }
}
ok: [host2] => {
    "find_hosts_output": {
        "changed": true,
        "cmd": "find hosts",
        "delta": "0:00:00.003155",
        "end": "2022-04-01 19:34:59.763075",
        "failed": false,
        "rc": 0,
        "start": "2022-04-01 19:34:59.759920",
        "stderr": "",
        "stderr_lines": [],
        "stdout": "hosts",
        "stdout_lines": [
            "hosts"
        ]
    }
}

PLAY RECAP **********************************************************************************************
host1                      : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
host2                      : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

A powerful pattern is to combine registered variables with conditionals to create tasks that will only be executed when certain custom conditions are true.

```
- name: Example Registered Variables Conditionals
  hosts: all
```

```
    - name: Check if hosts file contains the word "localhost"
      debug:
        msg: "/etc/hosts file contains the word localhost"
      when: hosts_contents.stdout.find("localhost") != -1
        var: find_hosts_output
```

Here, we registered in the variable *hosts_contents* the contents of /etc/hosts file, and we execute the second task only if the file contains the word *localhost*.

```
➜  ansible-variables git:(master) ✗ ansible-playbook example-register-variable-conditionals.yml

PLAY [Example Registered Variables Conditionals] ********************************************************

TASK [Gathering Facts] **********************************************************************************
ok: [host1]
ok: [host2]

TASK [Register an example variable] *********************************************************************
changed: [host1]
changed: [host2]

TASK [Check if hosts file contains the word "localhost"] ************************************************
ok: [host1] => {
    "msg": "/etc/hosts file contains the word localhost"
}
ok: [host2] => {
    "msg": "/etc/hosts file contains the word localhost"
}

PLAY RECAP **********************************************************************************************
host1                      : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
host2                      : ok=3    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Since registered variables are stored in memory, it's not possible to use them in future plays, and they are only available for the current playbook run.

- 10 Ways to Improve Your Infrastructure as Code
- Common Infrastructure Challenges and How to Solve Them
- 44 Ansible Best Practices to Follow

Share Variables with YAML Anchors and Aliases

When we want to reuse and share variables, we can leverage *YAML anchors and aliases*. They provide us with great flexibility in handling shared variables and help us reduce the repetition of data. Learn more in our Complete YAML Guide.

Anchors are defined with **&**, and then referenced with an *alias* denoted with *****. Let's go and check a hands-on example in a playbook.

```
- name: Example Anchors and Aliases
  hosts: all
  become: yes
  vars:
    user_groups: &user_groups
     - devs
     - support
    user_1:
        user_info: &user_info
            name: bob
            groups: *user_groups
            state: present
            create_home: yes
    user_2:
        user_info:
```

```
            name: jessica
            groups: support

  tasks:
  - name: Add several groups
    ansible.builtin.group:
      name: "{{ item }}"
      state: present
    loop: "{{ user_groups }}"

  - name: Add several users
    ansible.builtin.user:
      <<: *user_info
      name: "{{ item.user_info.name }}"
      groups: "{{ item.user_info.groups }}"
    loop:
      - "{{ user_1 }}"
      - "{{ user_2 }}"
      - "{{ user_3 }}"
```

Here, since some options are shared between users, instead of rewriting the same values, we share the common ones with the anchor *&user_info*. For every subsequent user declaration, we use the alias **user_info* to avoid repeating ourselves as much as possible.

The values for *state* and *create_home* are the same for all the users, while *name* and *groups* are replaced using the merge operator <<.

Similarly, we reuse the *user_groups* declaration in the definition of the *user_info* anchor. This way, we don't have to type the same groups again for *user_2* while we still have the flexibility to override the groups, as we do for *user_3*.

Variable Scope

Ansible provides many options on setting variables, and the ultimate decision on where to set them lies with us based on the scope we would like them to have. Conceptually, there are three main options available for scoping variables.

First, we have the **global** scope where the values are set for all hosts. This can be defined by the Ansible configuration, environment variables, and command line.

We set values for a particular host or group of hosts using the **host** scope. For example, there is an option to define some variables per host in the *inventory* file.

Lastly, we have the **play** scope, where values are set for all hosts in the context of a play. An example would be the *vars* section we have seen in previous examples in each playbook.

Variable Setting Options & Precedence

Variables can be defined with Ansible in many different places. There are options to set variables in playbooks, roles, inventory, var files, and command line. Let's go and explore some of these options.

As we have previously seen, the most straightforward way is to define variables in a play with the **vars** section.

```
- name: Set variables in a play
  hosts: all
  vars:
    version: 12.7.1
```

Another option is to define variables in the **inventory** file. We can set variables per host or set shared variables for groups. This example defines a different *ansible user* to connect for each host as a **host variable** and the same *HTTP port* for all web servers as a **group variable**.

```
[webservers]
webserver1 ansible_host=10.0.0.1 ansible_user=user1
webserver2 ansible_host=10.0.0.2 ansible_user=user2

[webservers:vars]
http_port=80
```

```
group_vars/databases
group_vars/webservers
host_vars/host1
host_vars/host2
```

Variables can also be set in *custom var files*. Let's check an example that uses variables from an external file and the *group_vars* and *host_vars* directories.

```
- name: Example External Variables file
  hosts: all
  vars_files:
    - ./vars/variables.yml

  tasks:
  - name: Print the value of variable docker_version
    debug:
      msg: "{{ docker_version}} "

  - name: Print the value of group variable http_port
    debug:
      msg: "{{ http_port}} "

  - name: Print the value of host variable app_version
    debug:
      msg: "{{ app_version}} "
```

The *group_vars/webservers* file:

```
http_port: 80
ansible_host: 127.0.0.1
ansible_user: vagrant
```

The *host_vars/host1* file:

```
app_version: 1.0.1
ansible_port: 2222
ansible_ssh_private_key_file: ./.vagrant/machines/host1/virtualbox/private_key
```

The *host_vars/host2* file:

```
app_version: 1.0.2
ansible_port: 2200
ansible_ssh_private_key_file: ./.vagrant/machines/host2/virtualbox/private_key
```

The inventory file contains a group named *webservers* that includes our two hosts, *host1* and *host2:*

If we run this playbook, we notice the same value is used in both hosts for the group variable *http_port* but a different one for the host variable *app_version.*

```
→  ansible-variables git:(master) ✗ ansible-playbook example-external-vars.yml

PLAY [Example External Variables file] *****************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [host1]
ok: [host2]

TASK [Print the value of variable docker_version] ******************************************************
ok: [host1] => {
    "msg": "20.10.12 "
}
ok: [host2] => {
    "msg": "20.10.12 "
}

TASK [Print the value of group variable http_port] *****************************************************
ok: [host1] => {
    "msg": "80 "
}
ok: [host2] => {
    "msg": "80 "
}

TASK [Print the value of host variable app_version] ****************************************************
ok: [host1] => {
    "msg": "1.0.1 "
}
ok: [host2] => {
    "msg": "1.0.2 "
}

PLAY RECAP *********************************************************************************************
host1                      : ok=4    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
host2                      : ok=4    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

A good use case for having separate variables files is that you can keep in them sensitive values without storing them in playbooks or source control systems.

```
ansible-playbook example-external-vars.yml --extra-vars "app_version=1.0.3"
```

Since variables can be set in multiple places, Ansible applies variable precedence to select the variable value according to some hierarchy. The general rule is that variables defined with a more explicit scope have higher priority.

For example, role defaults are overridden by mostly every other option. Variables are also flattened to each host before each play so all group and hosts variables are merged. Host variables have higher priority than group variables.

Explicit variables definitions like the *vars* directory or an *include_vars* task override variables from the inventory. Finally, extra vars defined at runtime always win precedence. For a complete list of options and their hierarchy, look at the official documentation Understanding variable precedence.

Where to Set Variables & Best Practices

Since Ansible provides a plethora of options to define variables, it might be a bit confusing to figure out the best way and place to set them. Let's go and check some common & best practices around setting variables that might help us better organize our Ansible projects.

- Always give descriptive and clear names to your variables. Taking a moment to properly think about how to name variables always pays off long-term.
- If there are default values for common variables, set them in **group_vars/all**
- Prefer setting group and host vars in group_vars and host_vars directories instead of in the inventory file.

- When you call roles, pass variables that you wish to override as parameters to make your plays easier to read.

```
roles:
       - role: example_role
         vars:
            example_var: 'example_string'
```

- You can always use **–extra-vars** or –**e** to override every other option.
- Don't store sensitive variables in your source code repository in plain text. You can leverage Ansible Vault in these cases.

In general, try to keep variable usage as simple as possible. You don't have to use all the existing options and spread variables definition all over the place because that makes debugging your Ansible projects difficult. Try to find a structure that suits your needs best and stick to it!

To learn more, check out the 44 Ansible Best Practices to Follow.

Key Points

In this article, we deep-dived into Ansible Variables and saw how we can define and use them in playbooks. Moreover, we explored different options for sharing, setting, and referencing them, along with some guidelines and best practices to make our Ansible journey easier.

by creating a free trial account.

Thank you for reading, and I hope you enjoyed this “Ansible Variables” article as much as I did.

Written by

Product

Documentation

How it works

Spacelift Tutorial

Pricing

Customer Case Studies

Integrations

Security

System Status

Product Updates

Company

About Us

Careers

Contact Sales

Partners

Learn