# Red Hat Enterprise Linux Automation with Ansible

FEEDBACK TRANSLATIONS CERTIFICATE OF ATTENDANCE

# Chapter 3. Managing Variables and Facts ☆



**Abstract**

| | |
|---|---|
| **Goal** | Write playbooks that use variables to simplify management of the playbook and facts to reference information about managed hosts. |

| | |
|---|---|
| **Objectives** | • Create and reference variables that affect particular hosts or host groups, the play, or the global environment, and describe how variable precedence works.<br>• Encrypt sensitive variables using Ansible Vault, and run playbooks that reference Vault-encrypted variable files.<br>• Reference data about managed hosts using Ansible facts, and configure custom facts on managed hosts. |
| **Sections** | • Managing Variables (and Guided Exercise)<br>• Managing Secrets (and Guided Exercise)<br>• Managing Facts (and Guided Exercise) |
| **Lab** | • Managing Variables and Facts |

# Managing Variables

## Objectives

After completing this section, you should be able to create and reference variables that affect particular hosts or host groups, the play, or the global environment, and describe how variable precedence works.

## Introduction to Ansible Variables

Ansible supports variables that can be used to store values that can then be reused throughout files in an Ansible project. This can simplify the creation and maintenance of a project and reduce the number of errors.

Variables provide a convenient way to manage dynamic values for a given environment in your Ansible project. Examples of values that variables might contain include:

- Users to create
- Packages to install
- Services to restart
- Files to remove
- Archives to retrieve from the internet

## Naming Variables

Variable names must start with a letter, and they can only contain letters, numbers, and underscores.

The following table illustrates the difference between invalid and valid variable names.

**Table 3.1. Examples of Invalid and Valid Ansible Variable Names**

| Invalid variable names | Valid variable names |
|---|---|
| `web server` | `web_server` |
| `remote.file` | `remote_file` |
| `1st file` | `file_1`<br>`file1` |
| `remoteserver$1` | `remote_server_1`<br>`remote_server1` |

## Defining Variables

Variables can be defined in a variety of places in an Ansible project. If a variable is set using the same name in two places, and those settings have different values, *precedence* determines which value is used.

You can set a variable that affects a group of hosts or only individual hosts. Some variables are *facts* that can be set by Ansible based on the configuration of a system. Other variables can be set inside the playbook, and affect one play in that playbook, or only one task in that play. You can also set *extra variables* on the `ansible-playbook` command line by using the `--extra-vars` or `-e` option and specifying those variables, and they override all other values for that variable name.

Here is a simplified list of ways to define a variable, ordered from lowest precedence to highest:

- Group variables defined in the inventory.
- Group variables defined in files in a `group_vars` subdirectory in the same directory as the inventory or the playbook.
- Host variables defined in the inventory.
- Host variables defined in files in a `host_vars` subdirectory in the same directory as the inventory or the playbook.
- Host facts, discovered at runtime.
- Play variables in the playbook (`vars` and `vars_files`).
- Task variables.
- Extra variables defined on the command line.

For example, a variable that is set to affect the `all` host group will be overridden by a variable that has the same name and is set to affect a single host.

One recommended practice is to choose globally unique variable names so you do not have to consider precedence rules. However, sometimes you might want to use precedence to cause different hosts or host groups to get different settings than your defaults.

If the same variable name is defined at more than one level, the level with the highest precedence wins. A narrow scope, such as a host variable or a task variable, takes precedence over a wider scope, such as a group variable or a play variable. Variables defined by the inventory are overridden by variables defined by the playbook. Extra variables defined on the command line with the `--extra-vars`, or `-e`, option have the highest precedence.

A detailed and more precise discussion of variable precedence is available in the Ansible documentation at Variable Precedence: Where Should I Put A Variable? (https://docs.ansible.com/ansible/2.9/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable).

## Variables in Playbooks

Variables play an important role in Ansible Playbooks because they ease the management of variable data in a playbook.

### Defining Variables in Playbooks

When writing playbooks, you can define your own variables and then invoke those values in a task. For example, a variable named `web_package` can be defined with a value of `httpd`. A task can then call the variable using the `yum` module to install the httpd package.

Playbook variables can be defined in multiple ways. One common method is to place a variable in a `vars` block at the beginning of a playbook:

```
- hosts: all
  vars:
    user: joe
    home: /home/joe
```

It is also possible to define playbook variables in external files. In this case, instead of using a `vars` block in the playbook, the `vars_files` directive may be used, followed by a list of names for external variable files relative to the location of the playbook:

```
- hosts: all
  vars_files:
    - vars/users.yml
```

The playbook variables are then defined in that file or those files in YAML format:

```
user: joe
home: /home/joe
```

### Using Variables in Playbooks

After variables have been declared, administrators can use the variables in tasks. Variables are referenced by placing the variable name in double curly braces ({{ }}). Ansible substitutes the variable with its value when the task is executed.

```
vars:
  user: joe

tasks:
  # This line will read: Creates the user joe
  - name: Creates the user {{ user }}
    user:
      # This line will create the user named Joe
      name: "{{ user }}"
```

**IMPORTANT**

When a variable is used as the first element to start a value, quotes are mandatory. This prevents Ansible from interpreting the variable reference as starting a YAML dictionary. The following message appears if quotes are missing:

```
yum:
     name: {{ service }}
            ^ here
We could be wrong, but this one looks like it might be an issue with
missing quotes.  Always quote template expression brackets when they
start a value. For instance:

    with_items:
      - {{ foo }}

Should be written as:

    with_items:
      - "{{ foo }}"
```

## Host Variables and Group Variables

Inventory variables that apply directly to hosts fall into two broad categories: *host variables* apply to a specific host, and *group variables* apply to all hosts in a host group or in a group of host groups. Host variables take precedence over group variables, but variables defined by a playbook take precedence over both.

One way to define host variables and group variables is to do it directly in the inventory file. This is an older approach and not the easiest to work with, but you might still see it used because it does put all the inventory information and variable settings for hosts and host groups in one file.

- Defining the `ansible_user` host variable for `demo.example.com`:

```
[servers]
demo.example.com  ansible_user=joe
```

- Defining the `user` group variable for the `servers` host group.

```
[servers]
demo1.example.com
demo2.example.com

[servers:vars]
user=joe
```

- Defining the `user` group variable for the `servers` group, which consists of two host groups each with two servers.

```
[servers1]
demo1.example.com
demo2.example.com

[servers2]
demo3.example.com
demo4.example.com

[servers:children]
servers1
servers2

[servers:vars]
user=joe
```

Some disadvantages of this approach are that it makes the inventory file more difficult to work with, it mixes information about hosts and variables in the same file, and uses an obsolete syntax.

### Using Directories to Populate Host and Group Variables

The preferred approach to defining variables for hosts and host groups is to create two directories, `group_vars` and `host_vars`, in the same working directory as the inventory file or playbook. These directories contain files defining group variables and host variables, respectively.

> **IMPORTANT**
>
> The recommended practice is to define inventory variables using `host_vars` and `group_vars` directories, and not to define them directly in the inventory files.

To define group variables for the `servers` group, you would create a YAML file named `group_vars/servers`, and then the contents of that file would set variables to values using the same syntax as in a playbook:

```
user: joe
```

Likewise, to define host variables for a particular host, create a file with a name matching the host in the `host_vars` directory to contain the host variables.

The following examples illustrate this approach in more detail. Consider a scenario where there are two data centers to manage and the data center hosts are defined in the `~/project/inventory` inventory file:

```
[admin@station project]$ cat ~/project/inventory
[datacenter1]
demo1.example.com
demo2.example.com

[datacenter2]
demo3.example.com
demo4.example.com

[datacenters:children]
datacenter1
datacenter2
```

- If you need to define a general value for all servers in both data centers, set a group variable for the `datacenters` host group:

```
[admin@station project]$ cat ~/project/group_vars/datacenters
package: httpd
```

- If the value to define varies for each data center, set a group variable for each data center host group:

```
[admin@station project]$ cat ~/project/group_vars/datacenter1
package: httpd
[admin@station project]$ cat ~/project/group_vars/datacenter2
package: apache
```

- If the value to be defined varies for each host in every data center, then define the variables in separate host variable files:

```
[admin@station project]$ cat ~/project/host_vars/demo1.example.com
package: httpd
[admin@station project]$ cat ~/project/host_vars/demo2.example.com
package: apache
[admin@station project]$ cat ~/project/host_vars/demo3.example.com
package: mariadb-server
[admin@station project]$ cat ~/project/host_vars/demo4.example.com
package: mysql-server
```

The directory structure for the example project, `project`, if it contained all the example files above, would appear as follows:

```
project
├── ansible.cfg
├── group_vars
│   ├── datacenters
│   ├── datacenters1
│   └── datacenters2
├── host_vars
│   ├── demo1.example.com
│   ├── demo2.example.com
│   ├── demo3.example.com
│   └── demo4.example.com
├── inventory
└── playbook.yml
```

> **NOTE**
>
> Ansible looks for `host_vars` and `group_vars` subdirectories relative to both the inventory and the playbook. If your inventory and your playbook happen to be in the same directory, this is simple and Ansible looks in that directory for those subdirectories. If your inventory and your playbook are in separate directories, then Ansible will look in both places for `host_vars` and `group_vars` subdirectories. The playbook subdirectories have higher precedence.

## Overriding Variables from the Command Line

Inventory variables are overridden by variables set in a playbook, but both kinds of variables may be overridden through arguments passed to the `ansible` or `ansible-playbook` commands on the command line. Variables set on the command line are called *extra variables*.

Extra variables can be useful when you need to override the defined value for a variable for a one-off run of a playbook. For example:

```
[user@demo ~]$ ansible-playbook main.yml -e "package=apache"
```

## Using Arrays as Variables

Instead of assigning configuration data that relates to the same element (a list of packages, a list of services, a list of users, and so on), to multiple variables, administrators can use arrays. One consequence of this is that an array can be browsed.

For example, consider the following snippet:

```
user1_first_name: Bob
user1_last_name: Jones
user1_home_dir: /users/bjones
user2_first_name: Anne
user2_last_name: Cook
user2_home_dir: /users/acook
```

This could be rewritten as an array called `users`:

```
users:
  bjones:
    first_name: Bob
    last_name: Jones
    home_dir: /users/bjones
  acook:
    first_name: Anne
    last_name: Cook
    home_dir: /users/acook
```

You can then use the following variables to access user data:

```
# Returns 'Bob'
users.bjones.first_name

# Returns '/users/acook'
users.acook.home_dir
```

Because the variable is defined as a Python *dictionary*, an alternative syntax is available.

```
# Returns 'Bob'
users['bjones']['first_name']

# Returns '/users/acook'
users['acook']['home_dir']
```

> **IMPORTANT**
>
> The dot notation can cause problems if the key names are the same as names of Python methods or attributes, such as `discard`, `copy`, `add`, and so on. Using the brackets notation can help avoid conflicts and errors.
>
> Both syntaxes are valid, but to make troubleshooting easier, Red Hat recommends that you use one syntax consistently in all files throughout any given Ansible project.

## Capturing Command Output with Registered Variables

You can use the `register` statement to capture the output of a command. The output is saved into a temporary variable that can be used later in the playbook for either debugging purposes or to achieve something else, such as a particular configuration based on a command's output.

The following playbook demonstrates how to capture the output of a command for debugging purposes:

```
---
- name: Installs a package and prints the result
  hosts: all
  tasks:
    - name: Install the package
      yum:
        name: httpd
        state: installed
      register: install_result

    - debug:
        var: install_result
```

When you run the playbook, the `debug` module is used to dump the value of the `install_result` registered variable to the terminal.

```
[user@demo ~]$ ansible-playbook playbook.yml
PLAY [Installs a package and prints the result] ***************************

TASK [setup] *******************************************************************
ok: [demo.example.com]

TASK [Install the package] *****************************************************
ok: [demo.example.com]

TASK [debug] *******************************************************************
ok: [demo.example.com] => {
    "install_result": {
        "changed": false,
        "msg": "",
        "rc": 0,
        "results": [
            "httpd-2.4.6-40.el7.x86_64 providing httpd is already installed"
        ]
    }
}

PLAY RECAP *********************************************************************
demo.example.com    : ok=3    changed=0    unreachable=0    failed=0
```

## REFERENCES


Inventory — Ansible Documentation (https://docs.ansible.com/ansible/2.9/user_guide/intro_inventory.html)

Variables — Ansible Documentation (https://docs.ansible.com/ansible/2.9/user_guide/playbooks_variables.html)

Variable Precedence: Where Should I Put A Variable? (https://docs.ansible.com/ansible/2.9/user_guide/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable)

YAML Syntax — Ansible Documentation (https://docs.ansible.com/ansible/2.9/reference_appendices/YAMLSyntax.html)



➜ NEXT (/ROL/APP/COURSES/RH294-8.4/PAGES/CH03S02)