Chapter 5 - Ansible Playbooks - Beyond the Basics

The playbooks and simple playbook organization we used in the previous chapter cover many common use cases. When discussing the breadth of system administration needs, there are thousands more features of Ansible you need to know.

We'll cover how to run plays with more granularity, how to organize your tasks and playbooks for simplicity and usability, and other advanced playbook topics that will help you manage your infrastructure with even more confidence.

Handlers

In chapter 4, the Ubuntu LAMP server example used a simple handler to restart Apache, and certain tasks that affected Apache's configuration notified the handler with the option `notify: restart apache`:

```
handlers:
  - name: restart apache
    service: name=apache2 state=restarted

tasks:
  - name: Enable Apache rewrite module.
    apache2_module: name=rewrite state=present
    notify: restart apache
```

In some circumstances you may want to notify multiple handlers, or even have handlers notify additional handlers. Both are easy to do with Ansible. To notify multiple handlers from one task, use a list for the `notify` option:

```
- name: Rebuild application configuration.
  command: /opt/app/rebuild.sh
  notify:
    - restart apache
    - restart memcached
```

To have one handler notify another, add a `notify` option onto the handler—handlers are basically glorified tasks that can be called by the `notify` option, but since they act as tasks themselves, they can chain themselves to other handlers:

```
handlers:
  - name: restart apache
    service: name=apache2 state=restarted
    notify: restart memcached

  - name: restart memcached
    service: name=memcached state=restarted
```

There are a few other considerations when dealing with handlers:

- Handlers will only be run if a task notifies the handler; if a task that would've notified the handlers is skipped due to a `when` condition or something of the like, the handler will not be run.
- Handlers will run once, and only once, at the end of a play. If you absolutely need to override this behavior and run handlers in the middle of a playbook, you can use the `meta` module to do so (e.g. `- meta: flush_handlers`).
- If the play fails on a particular host (or all hosts) before handlers are notified, the handlers will never be run. If it's desirable to always run handlers, even after the playbook has failed, you can use the `meta` module as described above as a separate task in the playbook, or you use the command line flag `--force-handlers` when running your playbook. Handlers won't run on any hosts that became unreachable during the playbook's run.

Environment variables

Ansible allows you to work with environment variables in a variety of ways. First of all, if you need to set some environment variables for your remote user account, you can do that by adding lines to the remote user's `.bash_profile`, like so:

```
- name: Add an environment variable to the remote user's shell.
  lineinfile: "dest=~/.bash_profile regexp=^ENV_VAR= \
  line=ENV_VAR=value"
```

All subsequent tasks will then have access to this environment variable (remember, of course, only the `shell` module will understand shell commands that use environment variables!). To use an environment variable in further tasks, it's recommended you use a task's `register` option to store the environment variable in a variable Ansible can use later, for example:

```
 1  - name: Add an environment variable to the remote user's shell.
 2    lineinfile: "dest=~/.bash_profile regexp=^ENV_VAR= \
 3    line=ENV_VAR=value"
 4
 5  - name: Get the value of the environment variable we just added.
 6    shell: 'source ~/.bash_profile && echo $ENV_VAR'
 7    register: foo
 8
 9  - name: Print the value of the environment variable.
10    debug: msg="The variable is {{ foo.stdout }}"
```

We use `source ~/.bash_profile` in line `4` because Ansible needs to make sure it's using the latest environment configuration for the remote user. In some situations, the tasks all run over a persistent or quasi-cached SSH session, over which `$ENV_VAR` wouldn't yet be defined.

(This is also the first time the `debug` module has made an appearance. It will be explored more in-depth along with other debugging techniques later.).

Why ~/.bash_profile? There are many different places you can store environment variables, including .bashrc, .profile, and .bash_login in a user's home folder. In our case, since we want the environment variable to be available to Ansible, which runs a pseudo-TTY shell session, in which case .bash_profile is used to configure the environment. You can read more about shell session configuration and these dotfiles in *Configuring your login sessions with dotfiles*[56].

Linux will also read global environment variables added to /etc/environment, so you can add your variable there:

```
- name: Add a global environment variable.
  lineinfile: "dest=/etc/environment regexp=^ENV_VAR= \
  line=ENV_VAR=value"
  become: yes
```

In any case, it's pretty simple to manage environment variables on the server with lineinfile. If your application requires many environment variables (as is the case in many Java applications), you might consider using copy or template with a local file instead of using lineinfile with a large list of items.

Per-play environment variables

You can also set the environment for just one play, using the environment option for that play. As an example, let's say you need to set an http proxy for a certain file download. This can be done with:

```
- name: Download a file, using example-proxy as a proxy.
  get_url: url=http://www.example.com/file.tar.gz dest=~/Downloads/
  environment:
    http_proxy: http://example-proxy:80/
```

That could be rather cumbersome, though, especially if you have many tasks that require a proxy or some other environment variable. In this case, you can pass an environment in via a variable in your playbook's vars section (or via an included variables file), like so:

[56]http://mywiki.wooledge.org/DotFiles

```
vars:
  proxy_vars:
    http_proxy: http://example-proxy:80/
    https_proxy: https://example-proxy:443/
    [etc...]

tasks:
- name: Download a file, using example-proxy as a proxy.
  get_url: url=http://www.example.com/file.tar.gz dest=~/Downloads/
  environment: proxy_vars
```

If a proxy needs to be set system-wide (as is the case behind many corporate firewalls), I like to do so using the global `/etc/environment` file:

```
# In the 'vars' section of the playbook (set to 'absent' to disable pro\
xy):
proxy_state: present

# In the 'tasks' section of the playbook:
- name: Configure the proxy.
  lineinfile:
    dest: /etc/environment
    regexp: "{{ item.regexp }}"
    line: "{{ item.line }}"
    state: "{{ proxy_state }}"
  with_items:
    - regexp: "^http_proxy="
      line: "http_proxy=http://example-proxy:80/"
    - regexp: "^https_proxy="
      line: "https_proxy=https://example-proxy:443/"
    - regexp: "^ftp_proxy="
      line: "ftp_proxy=http://example-proxy:80/"
```

Doing it this way allows me to configure whether the proxy is enabled per-server (using the `proxy_state` variable), and with one play, set the http, https, and ftp proxies. You can use a similar kind of play for any other types of environment variables you need to set system-wide.

You can test remote environment variables using the `ansible` command: `ansible test -m shell -a 'echo $TEST'`. When doing so, be careful with your use of quotes and escaping—you might end up using double quotes where you meant to use single quotes, or vice-versa, and end up printing a local environment variable instead of one from the remote server!

Variables

Variables in Ansible work just like variables in most other systems. Variables always begin with a letter (`[A-Za-z]`), and can include any number of underscores (`_`) or numbers (`[0-9]`).

Valid variable names include `foo`, `foo_bar`, `foo_bar_5`, and `fooBar`, though the standard is to use all lowercase letters, and typically avoid numbers in variable names (no `camelCase` or `UpperCamelCase`).

Invalid variable names include `_foo`, `foo-bar`, `5_foo_bar`, `foo.bar` and `foo bar`.

In an inventory file, a variable's value is assigned using an equals sign, like so:

```
foo=bar
```

In a playbook or variables include file, a variable's value is assigned using a colon, like so:

```
foo: bar
```

Playbook Variables

There are many different ways you can define variables to use in tasks.

Variables can be passed in via the command line, when calling `ansible-playbook`, with the `--extra-vars` option:

```
ansible-playbook example.yml --extra-vars "foo=bar"
```

You can also pass in extra variables using quoted JSON, YAML, or even by passing a JSON or YAML file directly, like `--extra-vars "@even_more_vars.json"` or `--extra-vars "@even_more_vars.yml`, but at this point, you might be better off using one of the other methods below.

Variables may be included inline with the rest of a playbook, in a `vars` section:

```
---
- hosts: example
  vars:
    foo: bar
  tasks:
    # Prints "Variable 'foo' is set to bar".
    - debug: msg="Variable 'foo' is set to {{ foo }}"
```

Variables may also be included in a separate file, using the `vars_files` section:

```
---
# Main playbook file.
- hosts: example
  vars_files:
    - vars.yml
  tasks:
    - debug: msg="Variable 'foo' is set to {{ foo }}"
```

```
---
# Variables file 'vars.yml' in the same folder as the playbook.
foo: bar
```

Notice how the variables are all at the root level of the YAML file. They don't need to be under any kind of `vars` heading when they are included as a standalone file.

Variable files can also be imported conditionally. Say, for instance, you have one set of variables for your CentOS servers (where the Apache service is named `httpd`), and another for your Debian servers (where the Apache service is named `apache2`). In this case, you could use a conditional `vars_files` include:

```
---
- hosts: example
  vars_files:
    - "apache_default.yml"
    - "apache_{{ ansible_os_family }}.yml"
  tasks:
    - service: name={{ apache }} state=running
```

Then, add two files in the same folder as your example playbook, `apache_CentOS.yml`, and `apache_default.yml`. Define the variable `apache: httpd` in the CentOS file, and `apache: apache2` in the default file.

As long as you don't disable `gather_facts` (or if you run a `setup` task at some point to gather facts manually), Ansible stores the OS of the server in the variable `ansible_os_family`, and will include the vars file with the resulting name. If ansible can't find a file with that name, it will use the variables loaded from the first loaded file (`apache_default.yml`). So, on a Debian or Ubuntu server, Ansible would correctly use `apache2` as the service name, even though there is no `apache_Debian.yml` or `apache_Ubuntu.yml` file available.

Inventory variables

Variables may also be added via Ansible inventory files, either inline with a host definition, or after a group:

```
# Host-specific variables (defined inline).
[washington]
app1.example.com proxy_state=present
app2.example.com proxy_state=absent

# Variables defined for the entire group.
[washington:vars]
cdn_host=washington.static.example.com
api_version=3.0.1
```

If you need to define more than a few variables, especially variables that apply to more than one or two hosts, inventory files can be cumbersome. In fact, Ansible's documentation recommends *not* storing variables within the inventory. Instead, you can use `group_vars` and `host_vars` YAML variable files within a specific path, and Ansible will assign them to individual hosts and groups defined in your inventory.

For example, to apply a set of variables to the host `app1.example.com`, create a blank file named `app1.example.com` at the location `/etc/ansible/host_vars/app1.example.com`, and add variables as you would in an included `vars_files` YAML file:

```
---
foo: bar
baz: qux
```

To apply a set of variables to the entire `washington` group, create a similar file in the location `/etc/ansible/group_vars/washington` (substitute `washington` for whatever group name's variables you're defining).

You can also put these files (named the same way) in `host_vars` or `group_vars` directories in your playbook's directory. Ansible will use the variables defined in the inventory `/etc/ansible/[host|group]_vars` directory first (if the appropriate files exist), then it will use variables defined in the playbook directories.

Another alternative to using `host_vars` and `group_vars` is to use conditional variable file imports, as was mentioned above.

Registered Variables

There are many times that you will want to run a command, then use its return code, stderr, or stdout to determine whether to run a later task. For these situations, Ansible allows you to use `register` to store the output of a particular command in a variable at runtime.

In the previous chapter, we used `register` to get the output of the `forever list` command, then used the output to determine whether we needed to start our Node.js app:

```
- name: "Node: Check list of Node.js apps running."
  command: forever list
  register: forever_list
  changed_when: false

- name: "Node: Start example Node.js app."
  command: forever start {{ node_apps_location }}/app/app.js
  when: "forever_list.stdout.find(node_apps_location + \
'/app/app.js') == -1"
```

In that example, we used a string function built into Python (`find`) to search for the path to our app, and if it was not present, the Node.js app was started.

We will explore the use of `register` further later in this chapter.

Accessing Variables

Simple variables (gathered by Ansible, defined in inventory files, or defined in playbook or variable files) can be used as part of a task using syntax like `{{ variable }}`. For example:

```
- command: /opt/my-app/rebuild {{ my_environment }}
```

When the command is run, Ansible will substitute the contents of `my_environment` for `{{ my_environment }}`. So the resulting command would be something like `/opt/my-app/rebuild dev`.

Many variables you will use are structured as arrays (or 'lists'), and accessing the array `foo` would not give you enough information to be useful (except when passing in the array in a context where Ansible will use the entire array, like when using `with_items`).

If you define a list variable like so:

```
foo_list:
  - one
  - two
  - three
```

You could access the first item in that array with either of the following syntax:

```
foo[0]
foo|first
```

Note that the first line uses standard Python array access syntax ('retrieve the first (`0`-indexed) element of the array'), whereas the second line uses a convenient *filter* provided by Jinja. Either way is equally valid and useful, and it's really up to you whether you like the first or second technique.

For larger and more structured arrays (for example, when retrieving the IP address of the server using the facts Ansible gathers from your server), you can access any part of the array by drilling through the array keys, either using bracket (`[]`) or dot (`.`) syntax. For example, if you would like to retrieve the information about the `eth0` network interface, you could first take a look at the entire array using `debug` in your playbook:

```
# In your playbook.
tasks:
  - debug: var=ansible_eth0


TASK: [debug var=ansible_eth0] ***************************************
ok: [webserver] => {
    "ansible_eth0": {
        "active": true,
        "device": "eth0",
        "ipv4": {
            "address": "10.0.2.15",
            "netmask": "255.255.255.0",
            "network": "10.0.2.0"
        },
        "ipv6": [
            {
                "address": "fe80::a00:27ff:feb1:589a",
                "prefix": "64",
                "scope": "link"
            }
        ],
        "macaddress": "08:00:27:b1:58:9a",
        "module": "e1000",
        "mtu": 1500,
        "promisc": false,
        "type": "ether"
    }
}
```

Now that you know the overall structure of the variable, you can use either of the following techniques to retrieve only the IPv4 address of the server:

```
{{ ansible_eth0.ipv4.address }}
{{ ansible_eth0['ipv4']['address'] }}
```

Host and Group variables

Ansible conveniently lets you define or override variables on a per-host or per-group basis. As we learned earlier, your inventory file can define groups and hosts like so:

```
[group]
host1
host2
```

The simplest way to define variables on a per-host or per-group basis is to do so directly within the inventory file:

```
[group]
host1 admin_user=jane
host2 admin_user=jack
host3

[group:vars]
admin_user=john
```

In this case, Ansible will use the group default variable 'john' for `{{ admin_user }}`, but for `host1` and `host2`, the admin users defined alongside the hostname will be used.

This is convenient and works well when you need to define a variable or two per-host or per-group, but once you start getting into more involved playbooks, you might need to add a few (3+) host-specific variables. In these situations, you can define the variables in a different place to make maintenance and readability much easier.

Automatically-loaded `group_vars` and `host_vars`

Ansible will search within the same directory as your inventory file (or inside `/etc/ansible` if you're using the default inventory file at `/etc/ansible/hosts`) for two specific directories: `group_vars` and `host_vars`.

You can place YAML files inside these directories named after the group name or hostname defined in your inventory file. Continuing our example above, let's move the specific variables into place:

```
---
# File: /etc/ansible/group_vars/group
admin_user: john
```

```
---
# File: /etc/ansible/host_vars/host1
admin_user: jane
```

Even if you're using the default inventory file (or an inventory file outside of your playbook's root directory), Ansible will also use host and group variables files located within your playbook's own `group_vars` and `host_vars` directories. This is convenient when you want to package together your entire playbook and infrastructure configuration (including all host/group-specific configuration) into a source-control repository.

You can also define a `group_vars/all` file that would apply to *all* groups. Usually, though, it's best to provide defaults in your playbooks and roles (which will be discussed later).

Magic variables with host and group variables and information

If you ever need to retrieve a specific host's variables from another host, Ansible provides a magic `hostvars` variable containing all the defined host variables (from inventory files and any discovered YAML files inside `host_vars` directories).

```
# From any host, returns "jane".
{{ hostvars['host1']['admin_user'] }}
```

There are a variety of other variables Ansible provides that you may need to use from time to time:

- `groups`: A list of all group names in the inventory.
- `group_names`: A list of all the groups of which the *current* host is a part.
- `inventory_hostname`: The hostname of the current host, according to the *inventory* (this can differ from `ansible_hostname`, which is the hostname reported by the system).
- `inventory_hostname_short`: The first part of `inventory_hostname`, up to the first period.
- `play_hosts`: All hosts on which the current play will be run.

Please see Magic Variables, and How To Access Information About Other Hosts[57] in Ansible's official documentation for the latest information and further usage examples.

[57]http://docs.ansible.com/playbooks_variables.html#magic-variables-and-how-to-access-information-about-other-hosts

Facts (Variables derived from system information)

By default, whenever you run an Ansible playbook, Ansible first gathers information ("facts") about each host in the play. You may have noticed this whenever we ran playbooks in earlier chapters:

```
$ ansible-playbook playbook.yml

PLAY [group] ******************************************************

GATHERING FACTS ***************************************************
ok: [host1]
ok: [host2]
ok: [host3]
```

Facts can be extremely helpful when you're running playbooks; you can use gathered information like host IP addresses, CPU type, disk space, operating system information, and network interface information to change when certain tasks are run, or to change certain information used in configuration files.

To get a list of every gathered fact available, you can use the `ansible` command with the `setup` module:

```
$ ansible munin -m setup
munin.midwesternmac.com | success >> {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "167.88.120.81"
        ],
        "ansible_all_ipv6_addresses": [
            "2604:180::a302:9076",
[...]
```

If you don't need to use facts, and would like to save a few seconds per-host when running playbooks (this can be especially helpful when running an Ansible playbook against dozens or hundreds of servers), you can set `gather_facts: no` in your playbook:

```
- hosts: db
  gather_facts: no
```

Many of my own playbooks and roles use facts like `ansible_os_family`, `ansible_hostname`, and `ansible_memtotal_mb` to register new variables or in tandem with `when`, to determine whether to run certain tasks.

If you have Facter[58] or Ohai[59] installed on a remote host, Ansible will also include their gathered facts as well, prefixed by `facter_` and `ohai_`, respectively. If you're using Ansible in tandem with Puppet or Chef, and are already familiar with those system-information-gathering tools, you can conveniently use them within Ansible as well. If not, Ansible's Facts are usually sufficient for whatever you need to do, and can be made even more flexible through the use of Local Facts.

If you run a playbook against similar servers or virtual machines (e.g. all your servers are running the same OS, same hosting provider, etc.), facts are almost always consistent in their behavior. When running playbooks against a diverse set of hosts (for example, hosts with different OSes, virtualization stacks, or hosting providers), know that some facts may contain different information than you were expecting. For Server Check.in[60], I have servers from no less than five different hosting providers, running on vastly different hardware, so I am sure to monitor the output of my `ansible-playbook` runs for abnormalities, especially when adding new servers to the mix.

Local Facts (Facts.d)

Another way of defining host-specific facts is to place `.fact` file in a special directory on remote hosts, `/etc/ansible/facts.d/`. These files can be either JSON or INI files, or you could use executables that return JSON. As an example, create the file `/etc/ansible/facts.d/settings.fact` on a remote host, with the following contents:

[58]https://puppet.com/docs/puppet/latest/facter.html
[59]https://docs.chef.io/ohai/
[60]https://servercheck.in/

```
[users]
admin=jane,john
normal=jim
```

Next, use Ansible's `setup` module to display the new facts on the remote host:

```
$ ansible hostname -m setup -a "filter=ansible_local"
munin.midwesternmac.com | success >> {
    "ansible_facts": {
        "ansible_local": {
            "settings": {
                "users": {
                    "admin": "jane,john",
                    "normal": "jim"
                }
            }
        }
    },
    "changed": false
}
```

If you are using a playbook to provision a new server, and part of that playbook adds a local `.fact` file which generates local facts that are used later, you can explicitly tell Ansible to reload the local facts using a task like the following:

```
- name: Reload local facts.
  setup: filter=ansible_local
```

While it may be tempting to use local facts rather than host_vars or other variable definition methods, remember that it's often better to build your playbooks in a way that doesn't rely (or care about) specific details of individual hosts. Sometimes it is necessary to use local facts (especially if you are using executables in facts.d to define the facts based on changing local environments), but it's almost always better to keep configuration in a central repository, and move away from host-specific facts.

Note that `setup` module options (like `filter`) won't work on remote Windows hosts, as of this writing.

Ansible Vault - Keeping secrets secret

If you use Ansible to fully automate the provisioning and configuration of your servers, chances are you will need to use passwords or other sensitive data for some tasks, whether it's setting a default admin password, synchronizing a private key, or authenticating to a remote service.

Some projects store such data in a normal variables file, in version control with the rest of the playbook, but in this case, the data is easily accessed by anyone with a copy of the project. It's better to treat passwords and sensitive data specially, and there are two primary ways to do this:

1. Use a separate secret management service, such as Vault[61] by HashiCorp, Keywhiz[62] by Square, or a hosted service like AWS's Key Management Service[63] or Microsoft Azure's Key Vault[64].
2. Use Ansible Vault, which is built into Ansible and stores encrypted passwords and other sensitive data alongside the rest of your playbook.

For most projects, Ansible's built-in Vault is adequate, but if you need some of the more advanced features found in the other projects listed in option #1 above, Ansible Vault might be too limiting.

Ansible Vault works much like a real-world vault:

1. You take any YAML file you would normally have in your playbook (e.g. a variables file, host vars, group vars, role default vars, or even task includes!), and store it in the vault.
2. Ansible encrypts the vault ('closes the door'), using a key (a password you set).

[61]https://vaultproject.io/
[62]http://square.github.io/keywhiz/
[63]https://aws.amazon.com/kms/
[64]http://azure.microsoft.com/en-us/services/key-vault/

3. You store the key (your vault's password) separately from the playbook in a location only you control or can access.
4. You use the key to let Ansible decrypt the encrypted vault whenever you run your playbook.

Let's see how it works in practice. Here's a playbook that connects to a service's API, and requires a secure API key to do so:

```
---
- hosts: appserver

  vars_files:
    - vars/api_key.yml

  tasks:
    - name: Connect to service with our API key.
      command: connect_to_service
      environment:
        SERVICE_API_KEY: "{{ myapp_service_api_key }}"
```

The vars_file, which is stored alongside the playbook, in plain text, looks like:

```
---
myapp_service_api_key: "yJJvPqhqgxyPZMispRycaVMBmBWPqYDf3DFanPxAMAm4UZc\
w"
```

This is convenient, but it's not safe to store the API key in plain text. Even when running the playbook locally on an access-restricted computer, secrets should be encrypted. If you're running the playbook via a central server (e.g. using Ansible Tower or Jenkins), or if you have this playbook in a shared repository, it's even more important. *You* may follow best practices for physical and OS security, but can you guarantee *every* developer and sysadmin who has access to this file does the same?

For the best security, use Ansible Vault to encrypt the file. If you ever checked the original file into version control, it's also a good time to expire the old key and generate a new one, since the old key is part of the plaintext history of your project!

To encrypt the file with Vault, run:

```
$ ansible-vault encrypt api_key.yml
```

Enter a secure password for the file, and Ansible will encrypt it. If you open the file now, you should see something like:

```
$ANSIBLE_VAULT;1.1;AES256
653635363963663439383865313262396665353063663839616266613737616539303
530313663316264336133626266336537616463366465653862366231310a30633064
633234306335333739623661633132376235666563653161353239383664613433663
13031323035663162323738653562373835396134376535633 00a3263386336393866
376535646562336664303137346432313563373534373264363835303739366362393
639646137656633656630313933323464333563376662643336616534353234663332
656138326530366434313161363562333639383864333635333766316161383832383
831626166623762643230313436386339373437333830306438653833666364653164
66336131323237386332663634 37
```

Next time you run the playbook, you will need to provide the password you used for the vault so Ansible can decrypt the playbook in memory for the brief period in which it will be used. If you don't specify the password, you'll receive an error:

```
$ ansible-playbook test.yml
ERROR: A vault password must be specified to decrypt vars/api_key.yml
```

There are a number of ways you can provide the password, depending on how you run playbooks. Providing the password at playbook runtime works well when running a playbook interactively:

```
# Use --ask-vault-pass to supply the vault password at runtime.
$ ansible-playbook test.yml --ask-vault-pass
Vault password:
```

After supplying the password, Ansible decrypts the vault (in memory) and runs the playbook with the decrypted data.

You can edit the encrypted file with `ansible-vault edit`. You can also `rekey` a file (change its password), `create` a new file, `view` an existing file, or `decrypt` a file. All these commands work with one or multiple files (e.g. `ansible-vault create x.yml y.yml z.yml`).

For convenience, or for automated playbook runs (e.g. on a continuous integration server), you can supply vault passwords via a password file. Just like secure keys in your `~/.ssh` folder, you should treat these files carefully: never check them into source control, and set strict permissions (e.g. `600`) so only *you* can read or write this file. Create the file `~/.ansible/vault_pass.txt` with your password in it, set permissions to `600`, and tell Ansible the location of the file when you run the playbook:

```
# Use --vault-password-file to supply the password via file/script.
$ ansible-playbook test.yml --vault-password-file ~/.ansible/\
vault_pass.txt
```

You could also use an executable script (e.g. `~/.ansible/vault_pass.py` with execute permissions, `700`), as long as the script outputs a single line of text, the vault password.

You can make Ansible's Vault operations slightly faster by installing Python's `cryptography` library, with `pip install cryptography`.

Are you worried about the security of Vault-encrypted files? AES-256 encryption is extremely secure; it would take *billions of billions* of years to decrypt this single file, even if all of today's fastest supercomputer clusters were all put to the task 24x7. Of course, every vault is only as secure as its password, so make sure you use a secure password and store *it* securely!

More options and examples are available in the official documentation for Ansible Vault[65].

[65]http://docs.ansible.com/ansible/playbooks_vault.html

Variable Precedence

It should be rare that you would need to dig into the details of which variable is used when you define the same variable in five different places, but since there are odd occasions where this is the case, Ansible's documentation provides the following ranking:

1. `--extra-vars` passed in via the command line (these always win, no matter what).
2. Task-level vars (in a task block).
3. Block-level vars (for all tasks in a block).
4. Role vars (e.g. `[role]/vars/main.yml`) and vars from `include_vars` module.
5. Vars set via `set_facts` modules.
6. Vars set via `register` in a task.
7. Individual play-level vars: 1. `vars_files` 2. `vars_prompt` 3. `vars`
8. Host facts.
9. Playbook `host_vars`.
10. Playbook `group_vars`.
11. Inventory: 1. `host_vars` 2. `group_vars` 3. `vars`
12. Role default vars (e.g. `[role]/defaults/main.yml`).

After lots of experience building playbooks, roles, and managing inventories, you'll likely find the right mix of variable definition for your needs, but there are a few general things that will mitigate any pain in setting and overriding variables on a per-play, per-host, or per-run basis:

- Roles (to be discussed in the next chapter) should provide sane default values via the role's 'defaults' variables. These variables will be the fallback in case the variable is not defined anywhere else in the chain.
- Playbooks should rarely define variables (e.g. via `set_fact`), but rather, variables should be defined either in included `vars_files` or, less often, via inventory.
- Only truly host- or group-specific variables should be defined in host or group inventories.

- Dynamic and static inventory sources should contain a minimum of variables, especially as these variables are often less visible to those maintaining a particular playbook.
- Command line variables (`-e`) should be avoided when possible. One of the main use cases is when doing local testing or running one-off playbooks where you aren't worried about the maintainability or idempotence of the tasks you're running.

See Ansible's Variable Precedence[66] documentation for more detail and examples, especially if you use older versions of Ansible (since older versions were not as strict about the precedence).

If/then/when - Conditionals

Many tasks need only be run in certain circumstances. Some tasks use modules with built-in idempotence (as is the case when ensuring a yum or apt package is installed), and you usually don't need to define further conditional behaviors for these tasks.

However, there are many tasks—especially those using Ansible's `command` or `shell` modules—which require further input as to when they're supposed to run, whether they've changed anything after they've been run, or when they've failed to run.

We'll cover all the main conditionals behaviors you can apply to Ansible tasks, as well as how you can tell Ansible when a play has done something to a server or failed.

Jinja Expressions, Python built-ins, and Logic

Before discussing all the different uses of conditionals in Ansible, it's worthwhile to cover a small part of Jinja (the syntax Ansible uses both for templates and for conditionals), and available Python functions (often referred to as 'built-ins'). Ansible uses expressions and built-ins with `when`, `changed_when`, and `failed_when` so you can describe these things to Ansible with as much precision as possible.

[66]http://docs.ansible.com/playbooks_variables.html#variable-precedence-where-should-i-put-a-variable

Jinja allows the definition of literals like strings (`"string"`), integers (`42`), floats (`42.33`), lists (`[1, 2, 3]`), tuples (like lists, but can't be modified) dictionaries (`{key: value, key2: value2}`), and booleans (`true` or `false`).

Jinja also allows basic math operations, like addition, subtraction, multiplication and division, and comparisons (== for equality, `!=` for inequality, `>=` for greater than or equal to, etc.). Logical operators are `and`, `or`, and `not`, and you can group expressions by placing them within parenthesis.

If you're familiar with almost any programming language, you will probably pick up basic usage of Jinja expressions in Ansible very quickly.

For example:

```
# The following expressions evaluate to 'true':
1 in [1, 2, 3]
'see' in 'Can you see me?'
foo != bar
(1 < 2) and ('a' not in 'best')

# The following expressions evaluate to 'false':
4 in [1, 2, 3]
foo == bar
(foo != foo) or (a in [1, 2, 3])
```

Jinja also offers a helpful set of 'tests' you can use to test a given object. For example, if you define the variable `foo` for only a certain group of servers, but not others, you can use the expression `foo is defined` with a conditional to evaluate to 'true' if the variable is defined, or false if not.

There are many other checks you can perform as well, like `undefined` (the opposite of `defined`), `equalto` (works like ==), `even` (returns true if the variable is an even number), and `iterable` (if you can iterate over the object). We'll cover the full gamut later in the book, but for now, know that you can use Ansible conditionals with Jinja expressions to do some powerful things!

For the few cases where Jinja doesn't provide enough power and flexibility, you can invoke Python's built-in library functions (like `string.split`, `[number].is_-`

signed()) to manipulate variables and determine whether a given task should be run, resulted in a change, failed, etc.

As an example, I need to parse version strings from time to time, to find the major version of a particular project. Assuming the variable software_version is set to 4.6.1, I can get the major version by splitting the string on the . character, then using the first element of the array. I can check if the major version is 4 using when, and choose to run (or not run) a certain task:

```
- name: Do something only for version 4 of the software.
  [task here]
  when: software_version.split('.')[0] == '4'
```

It's generally best to stick with simpler Jinja filters and variables, but it's nice to be able to use Python when you're doing more advanced variable manipulation.

register

In Ansible, any play can 'register' a variable, and once registered, that variable will be available to all subsequent tasks. Registered variables work just like normal variables or host facts.

Many times, you may need the output (stdout or stderr) of a shell command, and you can get that in a variable using the following syntax:

```
- shell: my_command_here
  register: my_command_result
```

Later, you can access stdout (as a string) with my_command_result.stdout, and stderr with my_command_result.stderr.

Registered facts are very helpful for many types of tasks, and can be used both with conditionals (defining when and how a play runs), and in any part of the play. As an example, if you have a command that outputs a version number string like "10.0.4", and you register the output as version, you can use the string later when doing a code checkout by printing the variable {{ version.stdout }}.

If you want to see the different properties of a particular registered variable, you can run a playbook with `-v` to inspect play output. Usually, you'll get access to values like `changed` (whether the play resulted in a change), `delta` (the time it took to run the play), `stderr` and `stdout`, etc. Some Ansible modules (like `stat`) add much more data to the registered variable, so always inspect the output with `-v` if you need to see what's inside.

`when`

One of the most helpful extra keys you can add to a play is a `when` statement. Let's take a look at a simple use of `when`:

```
- yum: name=mysql-server state=present
  when: is_db_server
```

The above statement assumes you've defined the `is_db_server` variable as a boolean (`true` or `false`) earlier, and will run the play if the value is true, or skip the play when the value is false.

If you only define the `is_db_server` variable on database servers (meaning there are times when the variable may not be defined at all), you could run tasks conditionally like so:

```
- yum: name=mysql-server state=present
  when: (is_db_server is defined) and is_db_server
```

`when` is even more powerful if used in conjunction with variables registered by previous tasks. As an example, we want to check the status of a running application, and run a play only when that application reports it is 'ready' in its output:

```
- command: my-app --status
  register: myapp_result

- command: do-something-to-my-app
  when: "'ready' in myapp_result.stdout"
```

These examples are a little contrived, but they illustrate basic uses of `when` in your tasks. Here are some examples of uses of `when` in real-world playbooks:

```
# From our Node.js playbook - register a command's output, then see
# if the path to our app is in the output. Start the app if it's
# not present.
- command: forever list
  register: forever_list
- command: forever start /path/to/app/app.js
  when: "forever_list.stdout.find('/path/to/app/app.js') == -1"

# Run 'ping-hosts.sh' script if 'ping_hosts' variable is true.
- command: /usr/local/bin/ping-hosts.sh
  when: ping_hosts

# Run 'git-cleanup.sh' script if a branch we're interested in is
# missing from git's list of branches in our project.
- command: chdir=/path/to/project git branch
  register: git_branches
- command: /path/to/project/scripts/git-cleanup.sh
  when: "(is_app_server == true) and ('interesting-branch' not in \
  git_branches.stdout)"

# Downgrade PHP version if the current version contains '7.0'.
- shell: php --version
  register: php_version
- shell: yum -y downgrade php*
  when: "'7.0' in php_version.stdout"

# Copy a file to the remote server if the hosts file doesn't exist.
```

```
- stat: path=/etc/hosts
  register: hosts_file
- copy: src=path/to/local/file dest=/path/to/remote/file
  when: hosts_file.stat.exists == false
```

`changed_when` and `failed_when`

Just like `when`, you can use `changed_when` and `failed_when` to influence Ansible's reporting of when a certain task results in changes or failures.

It is difficult for Ansible to determine if a given command results in changes, so if you use the `command` or `shell` module without also using `changed_when`, Ansible will always report a change. Most Ansible modules report whether they resulted in changes correctly, but you can also override this behavior by invoking `changed_when` yourself.

When using PHP Composer as a command to install project dependencies, it's useful to know when Composer installed something, or when nothing changed. Here's an example:

```
- name: Install dependencies via Composer.
  command: "/usr/local/bin/composer global require phpunit/phpunit \
  --prefer-dist"
  register: composer
  changed_when: "'Nothing to install' not in composer.stdout"
```

You can see we used `register` to store the results of the command, then we checked whether a certain string was in the registered variable's stdout. Only when Composer doesn't do anything will it print "Nothing to install or update", so we use that string to tell Ansible if the task resulted in a change.

Many command-line utilities print results to stderr instead of stdout, so `failed_when` can be used to tell Ansible when a task has *actually* failed and is not just reporting its results in the wrong way. Here's an example where we need to parse the stderr of a Jenkins CLI command to see if Jenkins did, in fact, fail to perform the command we requested:

```
- name: Import a Jenkins job via CLI.
  shell: >
    java -jar /opt/jenkins-cli.jar -s http://localhost:8080/
    create-job "My Job" < /usr/local/my-job.xml
  register: import
  failed_when: "import.stderr and 'exists' not in import.stderr"
```

In this case, we only want Ansible to report a failure when the command returns an error, **and** that error doesn't contain 'exists'. It's debatable whether the command should report a job already exists via stderr, or just print the result to stdout… but it's easy to account for whatever the command does with Ansible!

`ignore_errors`

Sometimes there are commands that should be run always, and they often report errors. Or there are scripts you might run that output errors left and right, and the errors don't actually indicate a problem, but they're just annoying (and they cause your playbooks to stop executing).

For these situations, you can add `ignore_errors: true` to the task, and Ansible will remain blissfully unaware of any problems running a particular task. Be careful using this, though; it's usually best if you can find a way to work with and around the errors generated by tasks so playbooks *do* fail if there are actual problems.

Delegation, Local Actions, and Pauses

Some tasks, like sending a notification, communicating with load balancers, or making changes to DNS, networking, or monitoring servers, require Ansible to run the task on the host machine (running the playbook) or another host besides the one(s) being managed by the playbook. Ansible allows any task to be delegated to a particular host using `delegate_to`:

```
- name: Add server to Munin monitoring configuration.
  command: monitor-server webservers {{ inventory_hostname }}
  delegate_to: "{{ monitoring_master }}"
```

Delegation is often used to manage a server's participation in a load balancer or replication pool; you might either run a particular command locally (as in the example below), or you could use one of Ansible's built-in load balancer modules and `delegate_to` a specific load balancer host directly:

```
- name: Remove server from load balancer.
  command: remove-from-lb {{ inventory_hostname }}
  delegate_to: 127.0.0.1
```

If you're delegating a task to localhost, Ansible has a convenient shorthand you can use, `local_action`, instead of adding the entire `delegate_to` line:

```
- name: Remove server from load balancer.
  local_action: command remove-from-lb {{ inventory_hostname }}
```

Pausing playbook execution with `wait_for`

You might also use `local_action` in the middle of a playbook to wait for a freshly-booted server or application to start listening on a particular port:

```
- name: Wait for web server to start.
  local_action:
    module: wait_for
    host: "{{ inventory_hostname }}"
    port: "{{ webserver_port }}"
    delay: 10
    timeout: 300
    state: started
```

The above task waits until `webserver_port` is open on `inventory_hostname`, as checked from the host running the Ansible playbook, with a 5-minute timeout (and 10 seconds before the first check, and between checks).

`wait_for` can be used to pause your playbook execution to wait for many different things:

- Using `host` and `port`, wait a maximum of `timeout` seconds for the port to be available (or not).
- Using `path` (and `search_regex` if desired), wait a maximum of `timeout` seconds for the file to be present (or absent).
- Using `host` and `port` and `drained` for the `state` parameter, check if a given port has drained all it's active connections.
- Using `delay`, you can simply pause playbook execution for a given amount of time (in seconds).

Running an entire playbook locally

When running playbooks on the server or workstation where the tasks need to be run (e.g. self-provisioning), or when a playbook should be otherwise run on the same host as the `ansible-playbook` command is run, you can use `--connection=local` to speed up playbook execution by avoiding the SSH connection overhead.

As a quick example, here's a short playbook that you can run with the command `ansible-playbook test.yml --connection=local`:

```
---
- hosts: 127.0.0.1
  gather_facts: no

  tasks:
    - name: Check the current system date.
      command: date
      register: date

    - name: Print the current system date.
      debug: var=date.stdout
```

This playbook will run on localhost and output the current date in a debug message. It should run *very* fast (it took about .2 seconds on my Mac!) since it's running entirely over a local connection.

Running a playbook with `--connection=local` is also useful when you're either running a playbook with `--check` mode to verify configuration (e.g. on a cron job that emails you when changes are reported), or when testing playbooks on testing infrastructure (e.g. via Travis, Jenkins, or some other CI tool).

Prompts

Under rare circumstances, you may require the user to enter the value of a variable that will be used in the playbook. If the playbook requires a user's personal login information, or if you prompt for a version or other values that may change depending on who is running the playbook, or where it's being run, and if there's no other way this information can be configured (e.g. using environment variables, inventory variables, etc.), use `vars_prompt`.

As a simple example, you can request a user to enter a username and password that could be used to login to a network share:

```
---
- hosts: all

  vars_prompt:
    - name: share_user
      prompt: "What is your network username?"

    - name: share_pass
      prompt: "What is your network password?"
      private: yes
```

Before Ansible runs the play, Ansible prompts the user for a username and password, the latter's input being hidden on the command line for security purposes.

There are a few special options you can add to prompts:

- `private`: If set to `yes`, the user's input will be hidden on the command line.
- `default`: You can set a default value for the prompt, to save time for the end user.
- `encrypt` / `confirm` / `salt_size`: These values can be set for passwords so you can verify the entry (the user will have to enter the password twice if `confirm` is set to `yes`), and encrypt it using a salt (with the specified size and crypt scheme). See Ansible's Prompts[67] documentation for detailed information on prompted variable encryption.

Prompts are a simple way to gather user-specific information, but in most cases, you should avoid them unless absolutely necessary. It's preferable to use role or playbook variables, inventory variables, or even local environment variables, to maintain complete automation of the playbook run.

Tags

Tags allow you to run (or exclude) subsets of a playbook's tasks.

You can tag roles, included files, individual tasks, and even entire plays. The syntax is simple, and below are examples of the different ways you can add tags:

```
---
# You can apply tags to an entire play.
- hosts: webservers
  tags: deploy

  roles:
    # Tags applied to a role will be applied to tasks in the role.
    - { role: tomcat, tags: ['tomcat', 'app'] }

  tasks:
    - name: Notify on completion.
      local_action:
        module: osx_say
```

[67]http://docs.ansible.com/playbooks_prompts.html#prompts

```
        msg: "{{inventory_hostname}} is finished!"
        voice: Zarvox
      tags:
        - notifications
        - say

    - import_tasks: foo.yml
      tags: foo
```

Assuming we save the above playbook as `tags.yml`, you could run the command below to only run the `tomcat` role and the `Notify on completion` task:

```
$ ansible-playbook tags.yml --tags "tomcat,say"
```

If you want to exclude anything tagged with `notifications`, you can use `--skip-tags`.

```
$ ansible-playbook tags.yml --skip-tags "notifications"
```

This is incredibly handy if you have a decent tagging structure; when you want to only run a particular portion of a playbook, or one play in a series (or, alternatively, if you want to exclude a play or included tasks), then it's easy to do using `--tags` or `--skip-tags`.

There is one caveat when adding one or multiple tags using the `tags` option in a playbook: you can use the shorthand `tags: tagname` when adding just one tag, but if adding more than one tag, you have to use YAML's list syntax, for example:

```
# Shorthand list syntax.
tags: ['one', 'two', 'three']

# Explicit list syntax.
tags:
  - one
  - two
  - three

# Non-working example.
tags: one, two, three
```

In general, I tend to use tags for larger playbooks, especially with individual roles and plays, but unless I'm debugging a set of tasks, I generally avoid adding tags to individual tasks or includes (not adding tags everywhere reduces visual clutter). You will need to find a tagging style that suits your needs and lets you run (or *not* run) the specific parts of your playbooks you desire.

Blocks

Introduced in Ansible 2.0.0, Blocks allow you to group related tasks together and apply particular task parameters on the block level. They also allow you to handle errors inside the blocks in a way similar to most programming languages' exception handling.

Here's an example playbook that uses blocks with `when` to run group of tasks specific to one platform without `when` parameters on each task:

```
---
- hosts: web
  tasks:
    # Install and configure Apache on RHEL/CentOS hosts.
    - block:
        - yum: name=httpd state=present
        - template: src=httpd.conf.j2 dest=/etc/httpd/conf/httpd.conf
        - service: name=httpd state=started enabled=yes
      when: ansible_os_family == 'RedHat'
      become: yes

    # Install and configure Apache on Debian/Ubuntu hosts.
    - block:
        - apt: name=apache2 state=present
        - template: src=httpd.conf.j2 dest=/etc/apache2/apache2.conf
        - service: name=apache2 state=started enabled=yes
      when: ansible_os_family == 'Debian'
      become: yes
```

If you want to perform a series of tasks with one set of task parameters (e.g. `with_items`, `when`, or `become`) applied, blocks are quite handy.

Blocks are also useful if you want to be able to gracefully handle failures in certain tasks. There might be a task that connects your app to a monitoring service that's not essential for a deployment to succeed, so it would be better to gracefully handle a failure than to bail out of the entire deployment!

Here's how to use a block to gracefully handle task failures:

```
tasks:
  - block:
      - name: Script to connect the app to a monitoring service.
        script: monitoring-connect.sh
    rescue:
      - name: This will only run in case of an error in the block.
        debug: msg="There was an error in the block."
    always:
      - name: This will always run, no matter what.
        debug: msg="This always executes."
```

Tasks inside the `block` will be run first. If there is a failure in any task in `block`, tasks inside `rescue` will be run. The tasks inside `always` will always be run, whether or not there were failures in either `block` or `rescue`.

Blocks can be very helpful for building reliable playbooks, but just like exceptions in programming languages, `block`/`rescue`/`always` failure handling can over-complicate things. If it's easier to maintain idempotence using `failed_when` per-task to define acceptable failure conditions, or to structure your playbook in a different way, it may not be necessary to use `block`/`rescue`/`always`.

Summary

Playbooks are Ansible's primary means of automating infrastructure management. After reading this chapter, you should know how to use (and hopefully not abuse!) variables, inventories, handlers, conditionals, tags, and more.

The more you understand the fundamental components of a playbook, the more efficient you will be at building and expanding your infrastructure with Ansible.

```
 ____________________________________
/ Men have become the tools of their \
\ tools. (Henry David Thoreau)       /
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```