← PREVIOUS (/ROL/APP/COURSES/RH294-8.4/PAGES/CH02S08) → NEXT (/ROL/APP/COURSES/RH294-8.4/PAGES/CH02S10)

Implementing Multiple Plays ☆

Objectives

After completing this section, you should be able to write a playbook that uses multiple plays and per-play privilege escalation, and effectively use `ansible-doc` to learn how to use new modules to implement tasks for a play

Writing Multiple Plays

A playbook is a YAML file containing a list of one or more plays. Remember that a single play is an ordered list of tasks to execute against hosts selected from the inventory. Therefore, if a playbook contains multiple plays, each play may apply its tasks to a separate set of hosts.

This can be very useful when orchestrating a complex deployment which may involve different tasks on different hosts. You can write a playbook that runs one play against one set of hosts, and when that finishes runs another play against another set of hosts.

Writing a playbook that contains multiple plays is very straightforward. Each play in the playbook is written as a top-level list item in the playbook. Each play is a list item containing the usual play keywords.

The following example shows a simple playbook with two plays. The first play runs against `web.example.com`, and the second play runs against `database.example.com`.

```
---
# This is a simple playbook with two plays

- name: first play
  hosts: web.example.com
  tasks:
    - name: first task
      yum:
        name: httpd
        status: present

    - name: second task
      service:
        name: httpd
        enabled: true

- name: second play
  hosts: database.example.com
  tasks:
    - name: first task
      service:
        name: mariadb
        enabled: true
```

Remote Users and Privilege Escalation in Plays

Plays can use different remote users or privilege escalation settings for a play than what is specified by the defaults in the configuration file. These are set in the play itself at the same level as the `hosts` or `tasks` keywords.

User Attributes

Tasks in playbooks are normally executed through a network connection to the managed hosts. As with ad hoc commands, the user account used for task execution depends on various keywords in the Ansible configuration file, `/etc/ansible/ansible.cfg`. The user that runs the tasks can be defined by the `remote_user` keyword. However, if privilege escalation is enabled, other keywords such as `become_user` can also have an impact.

If the remote user defined in the Ansible configuration for task execution is not suitable, it can be overridden by using the `remote_user` keyword within a play.

```
remote_user: remoteuser
```

Privilege Escalation Attributes

Additional keywords are also available to define privilege escalation parameters from within a playbook. The `become` boolean keyword can be used to enable or disable privilege escalation regardless of how it is defined in the Ansible configuration file. It can take `yes` or `true` to enable privilege escalation, or `no` or `false` to disable it.

```
become: true
```

If privilege escalation is enabled, the `become_method` keyword can be used to define the privilege escalation method to use during a specific play. The example below specifies that `sudo` be used for privilege escalation.

```
become_method: sudo
```

Additionally, with privilege escalation enabled, the `become_user` keyword can define the user account to use for privilege escalation within the context of a specific play.

```
become_user: privileged_user
```

The following example demonstrates the use of these keywords in a play:

```
- name: /etc/hosts is up to date
  hosts: datacenter-west
  remote_user: automation
  become: yes

  tasks:
    - name: server.example.com in /etc/hosts
      lineinfile:
        path: /etc/hosts
        line: '192.0.2.42 server.example.com server'
        state: present
```

Finding Modules for Tasks

Module Documentation

The large number of modules packaged with Ansible provides administrators with many tools for common administrative tasks. Earlier in this course, we discussed the Ansible documentation website at http://docs.ansible.com (http://docs.ansible.com). The *Module Index* on the website is an easy way to browse the list of modules shipped with Ansible. For example, modules for user and service management can be found under *Systems Modules* and modules for database administration can be found under *Database Modules*.

For each module, the Ansible documentation website provides a summary of its functions and instructions on how each specific function can be invoked with options to the module. The documentation also provides useful examples that show you how to use each module and how to set their keywords in a task.

You have already worked with the `ansible-doc` command to look up information about modules installed on the local system. As a review, to see a list of the modules available on a control node, run the `ansible-doc -l` command. This displays a list of module names and a synopsis of their functions.

```
[student@workstation modules]$ ansible-doc -l
a10_server              Manage A10 Networks ... devices' server object.
a10_server_axapi3       Manage A10 Networks ... devices
a10_service_group       Manage A10 Networks ... devices' service groups.
a10_virtual_server      Manage A10 Networks ... devices' virtual servers.
...output omitted...
zfs_facts               Gather facts about ZFS datasets.
znode                   Create, ... and update znodes using ZooKeeper
zpool_facts             Gather facts about ZFS pools.
zypper                  Manage packages on SUSE and openSUSE
zypper_repository       Add and remove Zypper repositories
```

Use the `ansible-doc [module name]` command to display detailed documentation for a module. Like the Ansible documentation website, the command provides a synopsis of the module's function, details of its various options, and examples. The following example shows the documentation displayed for the `yum` module.

```
[student@workstation modules]$ ansible-doc yum
> YUM    (/usr/lib/python3.6/site-packages/ansible/modules/packaging/os/yum.py)

        Installs, upgrade, downgrades, removes, and lists packages and groups with the `yum' package manager. This modul
e only works on Python 2. If you require Python
        3 support see the [dnf] module.

  * This module is maintained by The Ansible Core Team
  * note: This module has a corresponding action plugin.

OPTIONS (= is mandatory):

- allow_downgrade
        Specify if the named package and version is allowed to downgrade a maybe already installed higher version of tha
t package. Note that setting
        allow_downgrade=True can make this module behave in a non-idempotent way. The task could end up with a set of pa
ckages that does not match the complete list of
        specified packages to install (because dependencies between the downgraded package and others can cause changes
 to the packages which were in the earlier
        transaction).
        [Default: no]
        type: bool
        version_added: 2.4

- autoremove
        If `yes', removes all "leaf" packages from the system that were originally installed as dependencies of user-ins
talled packages but which are no longer required
        by any such package. Should be used alone or when state is `absent'
        NOTE: This feature requires yum >= 3.4.3 (RHEL/CentOS 7+)
        [Default: no]
        type: bool
        version_added: 2.7

- bugfix
        If set to `yes', and `state=latest' then only installs updates that have been marked bugfix related.
        [Default: no]
        version_added: 2.6

- conf_file
        The remote yum configuration file to use for the transaction.
        [Default: (null)]
        version_added: 0.6

- disable_excludes
        Disable the excludes defined in YUM config files.
        If set to `all', disables all excludes.
        If set to `main', disable excludes defined in [main] in yum.conf.
        If set to `repoid', disable excludes defined for given repo id.
        [Default: (null)]
        version_added: 2.7

- disable_gpg_check
        Whether to disable the GPG checking of signatures of packages being installed. Has an effect only if state is `p
resent' or `latest'.
        [Default: no]
        type: bool
        version_added: 1.2

- disable_plugin
        `Plugin' name to disable for the install/update operation. The disabled plugins will not persist beyond the tran
saction.
        [Default: (null)]
        version_added: 2.5

- disablerepo
        `Repoid' of repositories to disable for the install/update operation. These repos will not persist beyond the tr
ansaction. When specifying multiple repos,
        separate them with a `","'.
        As of Ansible 2.7, this can alternatively be a list instead of `","' separated string
        [Default: (null)]
```

The `ansible-doc` command also offers the `-s` option, which produces example output that can serve as a model for how to use a particular module in a playbook. This output can serve as a starter template, which can be included in a playbook to implement the module for task execution. Comments are included in the output to remind administrators of the use of each option. The following example shows this output for the `yum` module.

```
[student@workstation ~]$ ansible-doc -s yum
- name: Manages packages with the `yum' package manager
  yum:
      allow_downgrade:       # Specify if the named package ...
      autoremove:            # If `yes', removes all "leaf" packages ...
      bugfix:                # If set to `yes', ...
      conf_file:             # The remote yum configuration file ...
      disable_excludes:      # Disable the excludes ...
      disable_gpg_check:     # Whether to disable the GPG ...
      disable_plugin:        # `Plugin' name to disable ...
      disablerepo:           # `Repoid' of repositories ...
      download_only:         # Only download the packages, ...
      enable_plugin:         # `Plugin' name to enable ...
      enablerepo:            # `Repoid' of repositories to enable ...
      exclude:               # Package name(s) to exclude ...
      installroot:           # Specifies an alternative installroot, ...
      list:                  # Package name to run ...
      name:                  # A package name or package specifier ...
      releasever:            # Specifies an alternative release ...
      security:              # If set to `yes', ...
      skip_broken:           # Skip packages with ...
      state:                 # Whether to install ... or remove ... a package.
      update_cache:          # Force yum to check if cache ...
      update_only:           # When using latest, only update ...
      use_backend:           # This module supports `yum' ...
      validate_certs:        # This only applies if using a https url ...
```

Module Maintenance

Ansible ships with a large number of modules that can be used for many tasks. The upstream community is very active, and these modules may be in different stages of development. The `ansible-doc` documentation for the module is expected to specify who maintains that module in the upstream Ansible community, and what its development status is. This is indicated in the `METADATA` section at the end of the output of `ansible-doc` for that module.

The `status` field records the development status of the module:

- `stableinterface`: The module's keywords are stable, and every effort will be made not to remove keywords or change their meaning.
- `preview`: The module is in technology preview, and might be unstable, its keywords might change, or it might require libraries or web services that are themselves subject to incompatible changes.
- `deprecated`: The module is deprecated, and will no longer be available in some future release.
- `removed`: The module has been removed from the release, but a stub exists for documentation purposes to help former users migrate to new modules.

> **NOTE**
>
> The `stableinterface` status only indicates that a module's interface is stable, it does not rate the module's code quality.

The `supported_by` field records who maintains the module in the upstream Ansible community. Possible values are:

- `core`: Maintained by the "core" Ansible developers upstream, and always included with Ansible.
- `curated`: Modules submitted and maintained by partners or companies in the community. Maintainers of these modules must watch for any issues reported or pull requests raised against the module. Upstream "core" developers review proposed changes to curated modules after the community maintainers have approved the changes. Core committers also ensure that any issues with these modules due to changes in the Ansible engine are remediated. These modules are currently included with Ansible, but might be packaged separately at some point in the future.

- `community`: Modules not supported by the core upstream developers, partners, or companies, but maintained entirely by the general open source community. Modules in this category are still fully usable, but the response rate to issues is purely up to the community. These modules are also currently included with Ansible, but will be packaged separately at some point in the future.

The upstream Ansible community has an issue tracker for Ansible and its integrated modules at https://github.com/ansible/ansible/issues (https://github.com/ansible/ansible/issues).

Sometimes, a module does not exist for something you want to do. As an end user, you can also write your own private modules, or get modules from a third party. Ansible searches for custom modules in the location specified by the `ANSIBLE_LIBRARY` environment variable, or if that is not set, by a `library` keyword in the current Ansible configuration file. Ansible also searches for custom modules in the `./library` directory relative to the playbook currently being run.

```
library = /usr/share/my_modules
```

Information on writing modules is beyond the scope of this course. Documentation on how to do this is available at https://docs.ansible.com/ansible/2.9/dev_guide/developing_modules.html (https://docs.ansible.com/ansible/2.9/dev_guide/developing_modules.html).

IMPORTANT

Use the `ansible-doc` command to find and learn how to use modules for your tasks.

When possible, try to avoid the `command`, `shell`, and `raw` modules in playbooks, even though they might seem simple to use. Because these take arbitrary commands, it is very easy to write non-idempotent playbooks with these modules.

For example, the following task using the `shell` module is not idempotent. Every time the play is run, it rewrites `/etc/resolv.conf` even if it already consists of the line `nameserver 192.0.2.1`.

```
- name: Non-idempotent approach with shell module
  shell: echo "nameserver 192.0.2.1" > /etc/resolv.conf
```

There are several ways to write tasks using the `shell` module idempotently, and sometimes making those changes and using `shell` is the best approach. A quicker solution may be to use `ansible-doc` to discover the `copy` module and use that to get the desired effect.

The following example does not rewrite the `/etc/resolv.conf` file if it already consists of the correct content:

```
- name: Idempotent approach with copy module
  copy:
    dest: /etc/resolv.conf
    content: "nameserver 192.0.2.1\n"
```

The `copy` module tests to see if the state has already been met, and if so, it makes no changes. The `shell` module allows a lot of flexibility, but also requires more attention to ensure that it runs idempotently.

Idempotent playbooks can be run repeatedly to ensure systems are in a particular state without disrupting those systems if they already are.

Playbook Syntax Variations

The last part of this chapter investigates some variations of YAML or Ansible Playbook syntax that you might encounter.

YAML Comments

Comments can also be used to aid readability. In YAML, everything to the right of the number or hash symbol (#) is a comment. If there is content to the left of the comment, precede the number symbol with a space.

```
# This is a YAML comment
```

```
some data # This is also a YAML comment
```

YAML Strings

Strings in YAML do not normally need to be put in quotation marks even if there are spaces contained in the string. You can enclose strings in either double quotes or single quotes.

```
this is a string
```

```
'this is another string'
```

```
"this is yet another a string"
```

There are two ways to write multiline strings. You can use the vertical bar (|) character to denote that newline characters within the string are to be preserved.

```
include_newlines: |
        Example Company
        123 Main Street
        Atlanta, GA 30303
```

You can also write multiline strings using the greater-than (>) character to indicate that newline characters are to be converted to spaces and that leading white spaces in the lines are to be removed. This method is often used to break long strings at space characters so that they can span multiple lines for better readability.

```
fold_newlines: >
        This is an example
        of a long string,
        that will become
        a single sentence once folded.
```

YAML Dictionaries

You have seen collections of key-value pairs written as an indented block, as follows:

```
  name: svcrole
  svcservice: httpd
  svcport: 80
```

Dictionaries can also be written in an inline block format enclosed in curly braces, as follows:

```
  {name: svcrole, svcservice: httpd, svcport: 80}
```

In most cases the inline block format should be avoided because it is harder to read. However, there is at least one situation in which it is more commonly used. The use of *roles* is discussed later in this course. When a playbook includes a list of roles, it is more common to use this syntax to make it easier to distinguish roles included in a play from the variables being passed to a role.

YAML Lists

You have also seen lists written with the normal single-dash syntax:

```
  hosts:
    - servera
    - serverb
    - serverc
```

Lists also have an inline format enclosed in square braces, as follows:

```
hosts: [servera, serverb, serverc]
```

You should avoid this syntax because it is usually harder to read.

Obsolete key=value Playbook Shorthand

Some playbooks might use an older shorthand method to define tasks by putting the key-value pairs for the module on the same line as the module name. For example, you might see this syntax:

```
tasks:
  - name: shorthand form
    service: name=httpd enabled=true state=started
```

Normally you would write the same task as follows:

```
tasks:
  - name: normal form
    service:
      name: httpd
      enabled: true
      state: started
```

You should generally avoid the shorthand form and use the normal form.

The normal form has more lines, but it is easier to work with. The task's keywords are stacked vertically and easier to differentiate. Your eyes can run straight down the play with less left-to-right motion. Also, the normal syntax is native YAML; the shorthand form is not. Syntax highlighting tools in modern text editors can help you more effectively if you use the normal format than if you use the shorthand format.

You might see this syntax in documentation and older playbooks from other people, and the syntax does still function.

REFERENCES

`ansible-playbook`(1) and `ansible-doc`(1) man pages

Intro to Playbooks — Ansible Documentation (https://docs.ansible.com/ansible/2.9/user_guide/playbooks_intro.html)

Playbooks — Ansible Documentation (https://docs.ansible.com/ansible/2.9/user_guide/playbooks.html)

Developing Modules — Ansible Documentation (https://docs.ansible.com/ansible/2.9/dev_guide/developing_modules.html)

Module Support — Ansible Documentation (https://docs.ansible.com/ansible/2.9/user_guide/modules_support.html)

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

← PREVIOUS (/ROL/APP/COURSES/RH294-8.4/PAGES/CH02S08) → NEXT (/ROL/APP/COURSES/RH294-8.4/PAGES/CH02S10)