Red Hat
Training and
Certification

# Instructor Demos for Ansible – Custom Workshop

The instructor and students participating in this Ansible workshop will use demo playbooks and commands found in a public Git repository. To use the demo playbooks and commands, deploy virtual machines for the current RH294 lab environment. Then, as the student user on the workstation virtual machine, run the following commands to clone the Git repository and access the demo playbooks and commands.

```
[student@workstation ~]$ mkdir git-repos && cd git-repos
```

```
[student@workstation ~]$ git clone https://github.com/jessedscott/ansible_demo.git
```

```
[student@workstation ~]$ cd ansible_demo
```

This workshop uses ansible ad-hoc examples and will also use the ansible-playbook command from ansible-core to run playbooks. In addition to using modules provided by the ansible.builtin collection (which is included with the ansible-core installation), some playbook examples will use modules from the ansible.posix collection. To install the ansible.posix collection, run the following commands as the student user on the workstation virtual machine.:

```
[student@workstation ~]$ ansible-galaxy collection install ansible.posix
```

## What Is Ansible?

Ansible is an open source automation platform. It is a simple automation language that can accurately describe an IT application infrastructure in Ansible Playbooks. It is also an automation engine that runs Ansible Playbooks.

Ansible can manage powerful automation tasks and can adapt to many workflows and environments. At the same time, new users of Ansible can very quickly use it to become productive.

### Ansible Is Simple

Ansible Playbooks provide human-readable automation. This means that playbooks are automation tools that are also easy for humans to read, comprehend, and change. No special coding skills are required to write them. Playbooks execute tasks in order. The simplicity of playbook design makes them usable by every team, which allows people new to Ansible to get productive quickly.

### Ansible Is Powerful

You can use Ansible to deploy applications for configuration management, for workflow automation, and for network automation. You can use Ansible to orchestrate the entire application lifecycle.

### Ansible Is Agentless

Ansible is built around an agentless architecture. Typically, Ansible connects to the hosts it manages by using OpenSSH or WinRM and runs tasks, often (but not always) by pushing out small programs called Ansible modules to those hosts. These programs are used to put the system in a specific desired state. Any modules that are pushed are removed when Ansible has finished its tasks. You can start using Ansible almost immediately because no special agents need to be approved for use and then deployed to the managed hosts. Because there are no agents and no additional custom security infrastructure, Ansible is more efficient and more secure than other alternatives.

## Ansible Concepts and Architecture

The Ansible architecture consists of two types of machines: control nodes and managed hosts. Ansible is installed and run from a control node, and this machine also has copies of your Ansible project files.

Managed hosts are listed in an inventory, which also organizes those systems into groups for easier collective management. You can define the inventory statically in a text file, or dynamically using scripts that obtain group and host information from external sources.

Instead of writing complex scripts, Ansible users create high-level plays to ensure that a host or group of hosts is in a particular state. A play performs a series of tasks on the hosts, in the order specified by the play. These plays are expressed in YAML format in a text file. A file that contains one or more plays is called a playbook.

Each task runs a module, a small piece of code (written in Python, PowerShell, or some other language), with specific arguments. Each module is essentially a tool in your toolkit. Ansible ships with hundreds of useful modules that can perform a wide variety of automation tasks. They can act on system files, install software, or make API calls.

When used in a task, a module generally ensures that some particular aspect of the machine is in a particular state. For example, a task using one module might ensure that a file exists and has particular permissions and content. A task using a different module might ensure that a particular file system is mounted. If the system is not in that state, the task should put it in that state, or do nothing. If a task fails, the default Ansible behavior is to abort the rest of the playbook for the hosts that had a failure and continue with the remaining hosts.

Tasks, plays, and playbooks are designed to be idempotent. This means that you can safely run a playbook on the same hosts multiple times. When your systems are in the correct state, the playbook makes no changes when you run it. Numerous modules are available that you can use to run arbitrary commands. However, you must use those modules with care to ensure that they run in an idempotent way.
Ansible also uses plug-ins. Plug-ins are code that you can add to Ansible to extend it and adapt it to new uses and platforms.

The Ansible architecture is agentless. Typically, when an administrator runs an Ansible playbook, the control node connects to the managed host by using SSH (by default) or WinRM. This means that you do not need to have an Ansible-specific agent installed on managed hosts, and do not need to permit any additional communication between the control node and managed hosts.

## Ansible Core

Ansible Core provides the fundamental functionality used to run Ansible Playbooks. It defines the automation language that is used to write Ansible Playbooks in YAML text files. It provides the key functions such as loops, conditionals, and other Ansible imperatives needed for automation code.
It also provides the framework and basic command-line tools to drive automation.

## Ansible Content Collections

Historically, Ansible provided a large number of modules as part of the core package; an approach referred to in the Ansible community as "batteries included". However, with the success and rapid growth of Ansible, the number of modules included with Ansible grew exponentially. This led to certain challenges with support, especially because users sometimes wanted to use earlier or later versions of modules than were packaged with a particular version of Ansible.

The upstream developers decided to reorganize most modules into separate Ansible Content

Collections made up of related modules, roles, and plug-ins that are supported by the same group of developers. Ansible Core itself is limited to a small set of modules provided by the ansible.builtin Ansible Content Collection, which is always part of Ansible Core.

Red Hat provides access to more than 120 certified content collections with a Red Hat Ansible Automation Platform 2 subscription. Many community-supported collections are also available on Ansible Galaxy.

## Control Nodes

Ansible is simple to install. The Ansible software only needs to be installed on the control node (or nodes) from which Ansible will be run. Hosts that are managed by Ansible do not need to have Ansible installed. This installation involves relatively few steps and has minimal requirements.

The control node should be a Linux or UNIX system. Microsoft Windows is not supported as a control node, although Windows systems can be managed hosts.

Python 3 (version 3.5 or later) or Python 2 (version 2.7 or later) needs to be installed on the control node.

## Installing Ansible Core

In Red Hat Enterprise Linux 9, Ansible Core is available in the default AppStream repository. Because of this, we don't need to enable additional repositories. Ansible Core is provided as part of the rhel-9-for-x86_64-appstream-rpms channel for Red Hat. If you have a Red Hat Enterprise Linux 9 subscription, the installation procedure for Red Hat Ansible Core is as follows:

---

***Warning***

You do not need to run the next steps in the Red Hat RH294 classroom environment. These are provided for informational purposes only, and are useful for setting up your own Ansible control node on Red Hat Enterprise Linux 9.

---

1. Register your system to Red Hat Subscription Manager.

```
[root@host ~]# subscription-manager register
```

2. Install Red Hat Ansible Core.

```
[root@host ~]# dnf install ansible-core
```

### Managed Hosts

One of the benefits of Ansible is that managed hosts do not need to have a special agent installed. The Ansible control node connects to managed hosts using a standard network protocol to ensure that the systems are in the specified state.

Managed hosts might have some requirements depending on how the control node connects to them and what modules it will run on them. Linux and UNIX managed hosts need to have Python 2 (version 2.6 or later) or Python 3 (version 3.5 or later) installed for most modules to work.

On a Red Hat Enterprise Linux 9 based managed host, you can install the Python dependency by running the following command with root privileges.:

```
[root@host ~]# dnf install python3
```

## Ansible Static Inventory Files

### Defining the Inventory

An inventory defines a collection of hosts that Ansible will manage. These hosts can also be assigned to groups, which can be managed collectively. Groups can contain child groups, and hosts can be members of multiple groups. The inventory can also set variables that apply to the hosts and groups that it defines.

Host inventories can be defined in two different ways. A static host inventory can be defined by a text file. A dynamic host inventory can be generated by a script or other program as needed, using external information providers.

### Specifying Managed Hosts with a Static Inventory

A static inventory file is a text file that specifies the managed hosts that Ansible targets. You can write this file using a number of different formats, including INI-style or YAML. The INI-style format is very common and will be used for most examples in this course.

In its simplest form, an INI-style static inventory file is a list of host names or IP addresses of managed hosts, each on a single line:

```
web1.example.com
web2.example.com
db1.example.com
db2.example.com
192.0.2.42
```

Normally, however, you organize managed hosts into host groups. Host groups allow you to more effectively run Ansible against a collection of systems. In this case, each section starts with a host group name enclosed in square brackets ([]). This is followed by the host name or an IP address for each managed host in the group, each on a single line.

In the following example, the host inventory defines two host groups: webservers and db-servers.

```
[webservers]
web1.example.com
web2.example.com
192.0.2.42

[db-servers]
db1.example.com
db2.example.com
```

Hosts can be in multiple groups. In fact, recommended practice is to organize your hosts into multiple groups, possibly organized in different ways depending on the role of the host, its physical location, whether it is in production or not, and so on. This allows you to easily apply Ansible plays to specific hosts.

```
[webservers]
web1.example.com
web2.example.com
192.0.2.42

[db-servers]
db1.example.com
db2.example.com
```

```
[east-datacenter]
web1.example.com
db1.example.com

[west-datacenter]
web2.example.com
db2.example.com

[production]
web1.example.com
web2.example.com
db1.example.com
db2.example.com

[development]
192.0.2.42
```

**Important**

Two host groups always exist:

- The all host group contains every host explicitly listed in the inventory.
- The ungrouped host group contains every host explicitly listed in the inventory that is not a member of any other group.

## Defining Nested Groups

Ansible host inventories can include groups of host groups. This is accomplished by creating a host group name with the :children suffix. The following example creates a new group called north-america, which includes all hosts from the usa and canada groups.

```
[usa]
washington1.example.com
washington2.example.com

[canada]
ontario01.example.com
ontario02.example.com

[north-america:children]
canada
usa
```

A group can have both managed hosts and child groups as members. For example, in the previous inventory you could add a [north-america] section that has its own list of managed hosts. That list of hosts would be merged with the additional hosts that the north-america group inherits from its child groups.

## Simplifying Host Specifications with Ranges

You can specify ranges in the host names or IP addresses to simplify Ansible host inventories. You can specify either numeric or alphabetic ranges. Ranges have the following syntax:

```
[START:END]
```

Ranges match all values from START to END, inclusively. Consider the following examples:

- 192.168.[4:7].[0:255] matches all IPv4 addresses in the 192.168.4.0/22 network (192.168.4.0 through 192.168.7.255).
- server[01:20].example.com matches all hosts named server01.example.com through server20.example.com.
- [a:c].dns.example.com matches hosts named a.dns.example.com, b.dns.example.com, and c.dns.example.com.
- 2001:db8::[a:f] matches all IPv6 addresses from 2001:db8::a through 2001:db8::f.

If leading zeros are included in numeric ranges, they are used in the pattern. The second example above does not match server1.example.com but does match server07.example.com. To illustrate this, the following example uses ranges to simplify the [usa] and [canada] group definitions from the earlier example:

```
[usa]
washington[1:2].example.com

[canada]
ontario[01:02].example.com
```

# Ansible Ad Hoc Commands

## Running Ad Hoc Commands with Ansible

An ad hoc command is a way of executing a single Ansible task quickly, one that you do not need to save to run again later. They are simple, online operations that can be run without writing a playbook.

Ad hoc commands are useful for quick tests and changes. For example, you can use an ad hoc command to make sure that a certain line exists in the /etc/hosts file on a group of servers. You could use another ad hoc command to efficiently restart a service on many different machines, or to ensure that a particular software package is up-to-date.

Ad hoc commands are very useful for quickly performing simple tasks with Ansible. They do have their limits, and in general you will want to use Ansible Playbooks to realize the full power of Ansible. In many situations, however, ad hoc commands are exactly the tool you need to perform simple tasks quickly.

## Running Ad Hoc Commands

Use the ansible command to run ad hoc commands:

```
ansible host-pattern -m module [-a 'module arguments'] [-i inventory]
```

The host-pattern argument is used to specify the managed hosts on which the ad hoc command should be run. It could be a specific managed host or host group in the inventory. You have already seen this used in conjunction with the --list-hosts option, which shows you which hosts are matched by a particular host pattern. You have also already seen that you can use the -i option to specify a different inventory location to use than the default in the current Ansible configuration file.

The -m option takes as an argument the name of the module that Ansible should run on the targeted hosts. Modules are small programs that are executed to implement your task. Some modules need no additional information, but others need additional arguments to specify the details of their operation. The -a option takes a list of those arguments as a quoted string. One of the simplest ad hoc commands uses the ping module. This module does not do an ICMP ping, but checks to see if you can run Python-based modules on managed hosts. For

example, the following ad hoc command determines whether all managed hosts in the inventory can run standard modules:

```
[user@controlnode ~]$ ansible all -m ansible.builtin.ping
```

## Performing Tasks with Modules Using Ad Hoc Commands

Modules are the tools that ad hoc commands use to accomplish tasks. Ansible provides hundreds of modules which do different things. You can usually find a tested, special-purpose module that does what you need as part of the standard installation.

The ansible-doc -l command lists all modules installed on a system. You can use ansible-doc to view the documentation of particular modules by name, and find information about what arguments the modules take as options. For example, the following command displays documentation for the ping module:

```
[user@controlnode ~]$ ansible-doc ansible.builtin.ping
```

Most modules take arguments. You can find the list of arguments available for a module in the module's documentation. Ad hoc commands pass arguments to modules using the -a option. When no argument is needed, omit the -a option from the ad hoc command. If multiple arguments need to be specified, supply them as a quoted space-separated list.

For example, the following ad hoc command uses the user module to ensure that the newbie user exists and has UID 4000 on servera.lab.example.com:

```
[user@controlnode ~]$ ansible servera.lab.example.com \
-m ansible.builtin.user -a 'name=newbie uid=4000 state=present'
```

## Ansible Playbooks and Ad Hoc Commands

Ad hoc commands can run a single, simple task against a set of targeted hosts as a one-time command. The real power of Ansible, however, is in learning how to use playbooks to run multiple, complex tasks against a set of targeted hosts in an easily repeatable manner.
A play is an ordered set of tasks run against hosts selected from your inventory. A playbook is a text file containing a list of one or more plays to run in a specific order. Plays allow you to change a lengthy, complex set of manual administrative tasks into an easily repeatable routine with predictable and successful outcomes. In a playbook, you can save the sequence of tasks in a play into a human-readable and immediately runnable form. The tasks themselves,

because of the way in which they are written, document the steps needed to deploy your application or infrastructure.

## Writing and Using Ansible Playbooks

### Formatting an Ansible Playbook

To help you understand the format of a playbook, review this ad hoc command from a previous chapter:

```
[student@workstation ~]$ ansible servera.lab.example.com \
-m ansible.builtin.user -a "name=newbie uid=4000 state=present"
```

This can be rewritten as a single task play and saved in a playbook. The resulting playbook appears as follows:

```
---
- name: Configure important user consistently
  hosts: servera.lab.example.com
  tasks:
    - name: newbie exists with UID 4000
      ansible.builtin.user:
        name: newbie
        uid: 4000
        state: present
```

A playbook is a text file written in YAML format, and is normally saved with the extension yml. The playbook uses indentation with space characters to indicate the structure of its data. YAML does not place strict requirements on how many spaces are used for the indentation, but there are two basic rules.

- Data elements at the same level in the hierarchy (such as items in the same list) must have the same indentation.
- Items that are children of another item must be indented more than their parents.

You can also add blank lines for readability.

**Important**
Only the space character can be used for indentation; tab characters are not allowed.

If you use the vi text editor, you can apply some settings which might make it easier to edit your playbooks. For example, you can add the following line to your $HOME/.vimrc file, and when vi detects that you are editing a YAML file, it performs a 2-space indentation when you press the Tab key and autoindents subsequent lines.

```
autocmd FileType yaml setlocal ai ts=2 sw=2 et
```

A playbook begins with a line consisting of three dashes (---) as a start of document marker. It may end with three dots (...) as an end of document marker, although in practice this is often omitted.

In between those markers, the playbook is defined as a list of plays. An item in a YAML list starts with a single dash followed by a space. For example, a YAML list might appear as follows:

```
- apple
- orange
- grape
```

In the earlier playbook example, the line after --- begins with a dash and starts the first (and only) play in the list of plays.

The play itself is a collection of key-value pairs. Keys in the same play should have the same indentation. The following example shows a YAML snippet with three keys. The first two keys have simple values. The third has a list of three items as a value.

```
---
- name: just an example
  hosts: webservers
  tasks:
    - first
    - second
    - third
```

The original example play has three keys, name, hosts, and tasks, because these keys all have the same indentation. The first line of the example play starts with a dash and a space (indicating the play is the first item of a list), and then the first key, the name attribute. The name key associates an arbitrary string with the play as a label. This identifies what the play is for. The name key is optional, but is recommended because it helps to document your playbook. This is especially useful when a playbook contains multiple plays.

```
- name: Configure important user consistently
```

The second key in the play is a hosts attribute, which specifies the hosts against which the play's tasks are run. Like the argument for the ansible command, the hosts attribute takes a host pattern as a value, such as the names of managed hosts or groups in the inventory.

```
hosts: servera.lab.example.com
```

Finally, the last key in the play is the tasks attribute, whose value specifies a list of tasks to run for this play. This example has a single task, which runs the user module with specific arguments (to ensure user newbie exists and has UID 4000).

```
tasks:
  - name: newbie exists with UID 4000
    ansible.builtin.user:
      name: newbie
      uid: 4000
      state: present
```

The tasks attribute is the part of the play that actually lists, in order, the tasks to be run on the managed hosts. Each task in the list is itself a collection of key-value pairs.
In this example, the only task in the play has two keys:

- name is an optional label documenting the purpose of the task. It is a good idea to name all your tasks to help document the purpose of each step of the automation process.
- user is the module to run for this task. Its arguments are passed as a collection of key-value pairs, which are children of the module (name, uid, and state).

The following is another example of a tasks attribute with multiple tasks, using the service module to ensure that several network services are enabled to start at boot:
tasks:

```
- name: Web server is enabled
  ansible.builtin.service:
    name: httpd
    enabled: true

- name: NTP server is enabled
  ansible.builtin.service:
    name: chronyd
    enabled: true

- name: Postfix is enabled
  ansible.builtin.service:
    name: postfix
    enabled: true
```

---

**Important**
The order in which the plays and tasks are listed in a playbook is important, because Ansible runs them in the same order.

---

The playbooks you have seen so far are basic examples, and you will see more sophisticated examples of what you can do with plays and tasks as this workshop continues.

## Running Playbooks

The ansible-playbook command is used to run playbooks. The command is executed on the control node and the name of the playbook to be run is passed as an argument:

```
[student@workstation ~]$ ansible-playbook site.yml
```

When you run the playbook, output is generated to show the play and tasks being executed. The output also reports the results of each task executed.

The following example shows the contents of a simple playbook, and then the result of running it.

```
[student@workstation playdemo]$ cat webserver.yml
---
- name: play to setup web server
  hosts: servera.lab.example.com
  tasks:
    - name: latest httpd version installed
      ansible.builtin.dnf:
        name: httpd
        state: latest
...output omitted...
```

```
[student@workstation playdemo]$ ansible-playbook webserver.yml

PLAY [play to setup web server] ************************************

TASK [Gathering Facts] *********************************************
ok: [servera.lab.example.com]

TASK [latest httpd version installed] ******************************
changed: [servera.lab.example.com]

PLAY RECAP *********************************************************
servera.lab.example.com: ok=2  changed=1  unreachable=0  failed=0
```

Note that the value of the name key for each play and task is displayed when the playbook is run. (The Gathering Facts task is a special task that the setup module usually runs automatically at the start of a play.) For playbooks with multiple plays and tasks, setting name attributes makes it easier to monitor the progress of a playbook's execution.

You should also see that the latest httpd version installed task is changed for servera.lab.example.com. This means that the task changed something on that host to ensure its specification was met. In this case, it means that the httpd package probably was not installed or was not the latest version.

In general, tasks in Ansible Playbooks are idempotent, and it is safe to run a playbook multiple times. If the targeted managed hosts are already in the correct state, no changes should be made. For example, assume that the playbook from the previous example is run again:

```
[student@workstation playdemo]$ ansible-playbook webserver.yml
```

```
PLAY [play to setup web server] ***********************************

TASK [Gathering Facts] ********************************************
ok: [servera.lab.example.com]

TASK [latest httpd version installed] *****************************
ok: [servera.lab.example.com]

PLAY RECAP ********************************************************
servera.lab.example.com: ok=2 changed=0 unreachable=0 failed=0
```

This time, all tasks passed with status ok and no changes were reported.

## Syntax Verification

Prior to executing a playbook, it is good practice to perform a verification to ensure that the syntax of its contents is correct. The ansible-playbook command offers a --syntax-check option that you can use to verify the syntax of a playbook. The following example shows the successful syntax verification of a playbook.

```
[student@workstation ~]$ ansible-playbook –syntax-check webserver.yml

playbook: webserver.yml
```

When syntax verification fails, a syntax error is reported. The output also includes the approximate location of the syntax issue in the playbook. The following example shows the failed syntax verification of a playbook where the space separator is missing after the name attribute for the play.

```
student@workstation ~]$ ansible-playbook --syntax-check webserver.yml
ERROR! Syntax Error while loading YAML.

mapping values are not allowed in this context

The error appears to have been in ...output omitted... line 3, column 8, but may be elsewhere
in the file depending on the exact syntax problem.

The offending line appears to be:

 - name:play to setup web server
   hosts: servera.lab.example.com
```

```
  ^ here
```

## 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
      ansible.builtin.dnf:
        name: httpd
        state: present
    - name: second task
      ansible.builtin.service:
        name: httpd
        enabled: true

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

## 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.

| **Invalid variable names** | **Valid variable names** |
|---|---|
| web server | web_server |
| 1st file | file_1 |
| | file1 |
| remoteserver$1 | remote_server_1 |
| | remote_server1 |

## Defining Variables

Variables can be defined in a variety of places in an Ansible project. However, this can be simplified to three basic scope levels:

- Global scope: Variables set from the command line or Ansible configuration
- Play scope: Variables set in the play and related structures
- Host scope: Variables set on host groups and individual hosts by the inventory, fact gathering, or registered tasks

If the same variable name is defined at more than one level, the level with the highest precedence wins. A narrow scope takes precedence over a wider scope: variables defined by the inventory are overridden by variables defined by the playbook, which are overridden by variables defined on the command line.

## Variables in Playbooks

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

---

## Host Variables 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 directory. These directories contain files defining group variables and host variables, respectively.

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.

## 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"
```

## Task Iteration with Loops

Using loops saves administrators from the need to write multiple tasks that use the same module. For example, instead of writing five tasks to ensure five users exist, you can write one task that iterates over a list of five users to ensure they all exist.

Ansible supports iterating a task over a set of items using the loop keyword. You can configure loops to repeat a task using each item in a list, the contents of each of the files in a list, a generated sequence of numbers, or using more complicated structures. This section covers simple loops that iterate over a list of items. Consult the documentation for more advanced looping scenarios.

## Simple Loops

A simple loop iterates a task over a list of items. The loop keyword is added to the task, and takes as a value the list of items over which the task should be iterated. The loop variable item holds the value used during each iteration.

Consider the following snippet that uses the service module twice in order to ensure two network services are running:

```
- name: Postfix is running
  ansible.builtin.service:
    name: postfix
    state: started

- name: Dovecot is running
  ansible.builtin.service:
    name: dovecot
    state: started
```

These two tasks can be rewritten to use a simple loop so that only one task is needed to ensure both services are running:

```
tasks:

- name: Postfix and Dovecot are running
  ansible.builtin.service:
```

```
    name: "{{ item }}"
    state: started
  loop:
    - postfix
    - dovecot
```

The list used by loop can be provided by a variable. In the following example, the variable mail_services contains the list of services that need to be running.

```
vars:
  mail_services:
    - postfix
    - dovecot
tasks:
  - name: Postfix and Dovecot are running
    ansible.builtin.service:
      name: "{{ item }}"
      state: started
    loop: "{{ mail_services }}"
```

## Ansible Handlers

Ansible modules are designed to be idempotent. This means that in a properly written playbook, the playbook and its tasks can be run multiple times without changing the managed host unless they need to make a change to get the managed host to the desired state. However, sometimes when a task does make a change to the system, a further task may need to be run. For example, a change to a service's configuration file may then require that the service be reloaded so that the changed configuration takes effect.

Handlers are tasks that respond to a notification triggered by other tasks. Tasks only notify their handlers when the task changes something on a managed host. Each handler has a globally unique name and is triggered at the end of a block of tasks in a playbook. If no task notifies the handler by name then the handler will not run. If one or more tasks notify the handler, the handler will run exactly once after all other tasks in the play have completed.

Because handlers are tasks, administrators can use the same modules in handlers that they would use for any other task. Normally, handlers are used to reboot hosts and restart services.

Handlers can be considered as inactive tasks that only get triggered when explicitly invoked using a notify statement. The following snippet shows how the Apache server is only restarted by the restart apache handler when a configuration file is updated and notifies it:
tasks:

```
  - name: copy demo.example.conf configuration template
    ansible.builtin.template:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify:
      - restart apache

handlers:
  - name: restart apache
    ansible.builtin.service:
      name: httpd
      state: restarted
```

In the previous example, the restart apache handler triggers when notified by the template task that a change happened. A task may call more than one handler in its notify section. Ansible treats the notify statement as an array and iterates over the handler names:
tasks:

```
  - name: copy demo.example.conf configuration template
    ansible.builtin.template:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify:
      - restart mysql
      - restart apache
handlers:
  - name: restart mysql
    ansible.builtin.service:
      name: mariadb
      state: restarted
  - name: restart apache
    ansible.builtin.service:
      name: httpd
      state: restarted
```

## Describing the Benefits of Using Handlers

As discussed in the Ansible documentation, there are some important things to remember about using handlers:

- Handlers always run in the order specified by the handlers section of the play. They do not run in the order in which they are listed by notify statements in a task, or in the order in which tasks notify them.
- Handlers normally run after all other tasks in the play complete. A handler called by a task in the tasks part of the playbook will not run until all tasks under tasks have been processed. (There are some minor exceptions to this.)
- Handler names exist in a per-play namespace. If two handlers are incorrectly given the same name, only one will run.
- Even if more than one task notifies a handler, the handler only runs once. If no tasks notify it, a handler will not run.
- If a task that includes a notify statement does not report a changed result (for example, a package is already installed and the task reports ok), the handler is not notified. The handler is skipped unless another task notifies it. Ansible notifies handlers only if the task reports the changed status.

## References

Introduction to Ansible Ad Hoc Commands:
https://docs.ansible.com/ansible/latest/command_guide/intro_adhoc.html
Introduction to Writing and Running Playbooks:
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html
YAML Syntax for Ansible Playbooks:
https://docs.ansible.com/ansible/latest/reference_appendices/YAMLSyntax.html
Running Ansible Playbooks with ansible-playbook:
https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html#running-playbooks