Ansible:

It’s a Configuration Management tool.

Ansible compares the desired state with the current state, if there is a difference, ansible would update the current state with desired state.

Requirements for using Ansible:

* Hostname resolution to address managed hosts by name (ex: /etc/hosts)
* Python on all nodes must be present.
* SSH running on the managed nodes or servers.
* A dedicated user account with sudo privileges.
* Optional: SSH keys to make logging in easier. (for creating passwordless login(keygen command)) (good for RHCE exam) ( however, in the real world it’s dangerous).

Note: every user who is logging in, must have their own ssh-key.

* An inventory to identify managed nodes on the control node.
* An ‘ansible.cfg’ to specify default parameters. (for standard settings.)

Note: Issue: Permission denied (publickey,gssapi-keyex,gssapi-with-mic).

Might have to change file permissions.

Note: for Ansible usage use ‘ec2-user’ user.

Note: Examples of static inventory could be found in ‘**/etc/ansible/hosts’**. And its a default inventory file.

Settings for ansible: (Or Ansible Principles)

There are 3 levels to set up settings:

i) In the Command Line. (define ‘-b’ to escalate privileges) (highest priority)

ii) In Playbooks. ( define in yaml file, become=True)

iii) in Ansible.cfg ( Note: specific level settings will overwrite the generic settings. Default conf file is /etc/ansible/ansible.cfg which is generic. For specific levels, create a conf file in the project directory).( and if the settings has been given on command line it will overwrite the generic settings.)

**About ‘Ansible.cfg’:**

* Default file: /etc/ansible/ansible.cfg
* If an ‘ansible.cfg’ exists in the current project directory, that will be used and all settings in /etc/ansible/ansible.cfg ignored.
* If a playbook contains settings, they will override the settings from ansible.cfg
* if the settings has been given on the command line it will overwrite the generic settings. (and it has Highest priority).

Note: to verify which ‘ansible.cfg’ is being used. Run ‘Ansible --version’

To connect to Remote host:

Generate, keygen (public and private key) and ssh-copy-id (copy it to the remote host.)

Note: make sure to do these in exam or before going to exam watch a video of it.

**Escalating Privileges:**

Note: password-less escalation is OK on the RHCE exam, but insecure and not recommended in the real world.

To set-up password-less sudo, create a drop-in file in ‘/etc/sudoers.d/’ with the following content:

* Username ALL=(ALL) NOPASSWD: ALL

**Ansible-navigator**:

It is a command-line tool designed to enhance the user experience of running Ansible, a popular automation tool used for IT tasks like configuration management, application deployment, and task automation. Ansible-navigator aims to streamline the process of executing and managing Ansible playbooks, inventories, and other related tasks.

Ansible-navigator uses a container based execution environment.

Note: Using Execution environment, means that you don’t have to set up an ansible control host, but just run required content from the execution environment.

Generic conf file(generic settings): ~/.ansible-navigator.yml

If an ‘ansible-navigator.yml’ is found in the current project directory, this will have higher priority.

Module: (also referred to as task plugins or library plugins) are discrete units of code that can be used from the command line or in playbook tasks.

Module, shell is similar to command module but advantage of shell is, it allows to use the pipe( | ) and redirects (>>).

Note: FQCN = Fully Qualified Collection Names. (Ex: ansible.builtin.user) (recommended to use) and short name (ex: ansible1)

**Collections**: Collection is the new way to bundle ansible contents to make it more manageable.

Collections may have modules, plugins, roles.

Note: for modules help, use

Ansible-doc -t module\_name

Ansible-navigator doc -m stdout --pp module\_name. # stdout is for standard output.

Use ‘--help’ for usage.

**Installing Collections from different locations:**

* From Ansible Galaxy : ‘ansible-galaxy collection install my.collection -p collections’
* From URL: ‘ansible-galaxy collection install https:url.tar.gz -p collections’
* From github: ‘ansible-galaxy collection install [git@git.example.local](mailto:git@git.example.local):mygitaccount/mycollection.git -p collections’
* Tar file: ‘ ansible-galaxy collection install /my/collection.tar.gz -p collections’

Note: ‘-p’ specifies the path. And collection path can be mentioned in ‘ansible.cfg’ as ‘collections\_path: file\_path’

Note: to install roles and collections use galaxy.

**Installing collections using ‘requirements.yml’:**

File ‘requirements.yml’:

* Src : community.aws

Name: name\_collection

* Src: ansible.posix

Name: name\_collection

Then execute: ansible-galaxy collection install -r requrements.yml\_location -p path\_of collections.

**Idempotency**: Means, that no matter the current state of the managed node, running an ansible module always gives the same result.

Some modules are idempotent in nature and some are not (avoid using command, shell, raw) because they are not idempotent.

Note:If the current state already matches the desired state, the ansible module should not generate an error message.

NOte: In ansible you should always configure idempotent solutions.

Note: ansible doc is available in exam to access. Open browser and type. ‘docs.ansible.com’.

Ansible Playbooks:

Basically, It consists of a set of instructions that has to be executed on one or more servers. These instructions could be for installations of a software (or) Upgrading a software. And it can be written in YAML or JSON format.

And can be used in complex tasks and playbooks are idempotent.

Playbooks can help with dependencies .

Use only spaces (2 spaces). Tab doesn’t work in yaml.

Note: A good playbook is written to be idempotent. Idempotency is very important.

Note: Once you run a playbook it can’t be undone. To undo the things you have to write a separate playbook which will undo the things.

Playbook Errors:

Plays consist of tasks.

In a Play, if a first task fails it won’t execute the second task because the first task has failed.

Note: for the RHCE exam, you can use ansible and ansible-navigator. Both will give the same results. However, I prefer to use ansible ‘cause it has short commands and I'm familiar with it.

**Verifying the Playbook syntax:**

Specifically, checking the syntax: ‘ansible-playbook --syntax-check filename’.

You can use the verbosity. Add ‘-v’.

Note: we can **pass** the exam without you using the ansible-navigator once.

Note: ‘docs.ansible.com’. Document will be available in the exam.

Note: if a module is not present, try to download a collection from the ‘galaxy.ansible.com’.

And ‘ansible-galaxy collection install collectio\_name -p collectio\_path’.

Mention a line of collection\_path in ‘ansible.cfg’ and mention the collection path.

**Lesson 5: Working with Variables.**

**Variables:** It’s a label that is assigned to a specific value.

* Variables can be defined by admins at different levels.
* Different types of variables can be used and defined. (ex: string, int, Boolean)
* Variables can be defined in playbooks.
* It could be defined in inventory.
* Variables can be defined in ‘include file’.
* Variables can be mentioned as command line arguments.
* Vars\_prompt, can be useful to take user input. (which will make the playbook interactive.)
* Host variables: It consists of properties of the host which is specific to a host. And it is only defined in inventory.
* System variables: they are built-in variables and cannot be changed.

**Register**: It stores the results of a task in a variable.

**Ansible Facts**: Ansible Facts are system properties and they are stored in variables.. (and it refers to the current state of a managed system.)

**Variable precedence:** Variables can be set with different types of scope:

* Global scope: this is when a variable is set from inventory or the command line.
* Play scope: This is applied when it is set from a play.
* Host scope: This is applied when set in inventory or using a host variable inclusion file.
* Set\_fact module can be used to define variables and the advantage of it has got scope. Note: host must be the same, it won’t work with a different host.
* Ex:
* Set\_fact:
* Variable\_name: value
* Access this variable in different plays but make sure hosts are the same.
* **(Ansible Principles)**

There are 3 levels to set up settings:

i) In the Command Line. (define ‘-b’ to escalate privileges) (highest priority)

ii) In Playbooks. ( define in yaml file, become=True)

iii) in Ansible.cfg ( Note: specific level settings will overwrite the generic settings. Default conf file is /etc/ansible/ansible.cfg which is generic. For specific levels, create a conf file in the project directory).( and if the settings has been given on command line it will overwrite the generic settings.)

Using Variables:

To access or refer variables : {{ var\_name }}

Note: if the variable is the first element, using quotes is mandatory: “{{ var\_name }}”.

Including variables from file:

In the play header,

* Name: play for vars

Hosts: all

Vars\_files: # including file.

* file\_path

# access the variables.

Tasks:

* Name: vars

Debug:

Msg: ‘the message of vars {{ var\_name }}’

Note: if you came across a keyword and you aren’t aware of it, use

‘ ansible-doc -t keyword what\_keyword\_is’

Note: if there is no reference of variables in the playbook. Check if there is any group\_vars or host\_vars directory.

To define host\_vars, create a directory with the name ‘host\_vars’ and a file(hostname) that contains host variables. (Important: It has a playbook scope but the host must be the same.)

To define group\_vars, create a directory with the name ‘group\_vars’ and

a file(group\_name\_of\_servers) that contains a group variable. (Important: It has a playbook scope)

Note: it will automatically be picked up by ansible.

There might be chances you would encounter **system variables.**

Ex: In host\_vars/ansible1: (vars)

User: lisa

Groups: wheel # Here ‘Groups’ is a system variable. Check the manual of the user module.

When you access and execute these, output will be different than you would expect.

**Register**: It stores the results of a task in a variable.

Ex: - name: adding user

User:

Name: lisa

Register: task\_result.

* Name: using Register

Debug:

Var: task\_result # accessing the register(task\_results).

Ansible-Vault: It processes the sensitive information in a secure way.

Ex: sensitive info could be web-keys, passwords and more.

Encrypt a new file with: ansible-vault create file\_name. (Or)

Encrypt an existing file: ansible-vault encrypt file\_name. # I have to set the vault password.

Decrypt a file with: ansible-vault decrypt file\_name.

To use a encrypt file : ansible-playbook -- ask-vault-pass playbook\_name

It will prompt for a vault password.

Or give a vault password file ( it is a secure way, because you have a file(password file) in your home directory).

Ansible-playbook - - vault-password-file vault\_password\_file playbook\_name.

**Ansible Facts**: Ansible Facts are system properties and they are stored in variables.. (and it refers to the current state of a managed system.)

To disable the Gather\_Facts, use “gather\_facts: no” in the play header.

Note: gather\_facts could save time in Exam. however, ensure that there is no task related to facts.

Note: Facts are stored as Multi-tier variables. To access them use square braces (ex: ansible\_facts[‘date\_time’][‘date’] which is the **preferred** way. or dotted notation (Ex: ansible\_facts.date\_time.date).

They are useful to perform a task based on certain conditions

Note: TroubleShooting The Fact collection: (sometimes fact gathering will consume more time in order to overcome, make sure name resolution is there in remote machines.)

Mention the Control hostname(/etc/hosts) in a remote machine.(It’s not important for RHCE but it could save time in exams).

Dictionary(Hash): consists of keys and value pairs. Loop **doesn’t** work with it.

Array(lists) consist of a list of items. Loop can work with it.

Note: Recognizing Arrays(List) and Dictionaries(Hash), the list is always written between ‘[ ]’.

Dictionaries are written between ‘{ }’. More importantly, Dictionaries can be part of a list and lists can be a part of Dictionaries.

**Custom Facts:**

They are created in ‘ini or json’ format. They must be present in a managed host.

Ex: of ‘ini-format’

[custom\_label]

package = httpd

service = rsync

NOte: Filename extension must be (.fact) and must have a label.

Custom facts are created in ‘/etc/ansible/facts.d’. ( of all managed nodes)

**Understanding Variable scope:**

* Variables set in a playbook only exist in the specific play where they are defined.
* Variables can get a playbook scope by defining them as follows:

- Through the 'host var’ directory or ‘host group var’ directory. ( has playbook scope and variables can be accessible to all plays.)

- In inventory ( it’s an old way)

- By vars plugins ( not important for RHCE)

- Using modules like ‘set\_fact and include\_vars’. ( has a playbook scope but the host must be the same).

**Variable Precedence:**

Generally in Ansible, More specific will always win.

Below is a partial list of precedence, from lowest to highest.

* Role defaults.
* Inventory variables.
* Host facts.
* Play variables. (Play variables are defined directly within the playbook under the vars section.)
* Included play variables. (Variables included using the include\_vars module are loaded from external files or directories.)
* Task variables.
* Variables provided on the command line.

**Lesson 7:**

**Conditionals statements: (task controls)**

* **Loop**: allows you to loop over a list of items instead of calling the same task repeatedly.

Loop is a task modifier. (basically, used at the end of a task. And indentation level is equal).

Loop and item are the keywords in ansible.

Item: can be used to iterate over a list.

**NOTE:** when looping over the list use, **loop: “{{ list\_name }}”**

* **When**: is used for conditional task execution. (**NOTE:** when using ‘when’ don’t put double-quotes.)
* **Handlers**: Handlers are the tasks that only run if triggered by a task that has changed something.

NOte: Plugins and filters are not important for RHCE.

Example for loop\_1:

vars:

loop\_vars:

- sshd

- crond

tasks:

- name: task for starting service

service:

name: "{{ item }}"

state: started

loop: "{{ loop\_vars }}"

Example for loop\_2:

tasks:

- name: task for dict

user:

name: "{{ item.username }}"

group: "{{ item.groups\_ }}"

state: present

Loop: # its list of dictionaries.

- username: alexa

groups\_: wheel

- username: google

groups\_: wheel

**Note**: when using conditions, it is important to address(or check) the right variable type.

**Note**: when using ‘when statement’, variables are in **‘jinja2 template’**. So you don’t have to use double quotes or curly braces.

**Note**: Playbook variables, registered variables and facts can be used in conditions and make sure that tasks only run if specific conditions are true.

Example of when:

vars:

supported\_os:

- CentOS

- Ubuntu

- RedHat

my\_package: nmap

tasks:

- name: name of the task.

yum:

name: "{{ my\_package }}"

state: present

when: ansible\_distribution in supported\_os

**Multiple conditions:**

* ‘When’ can be used to test multiple conditions as well.
* Use ‘and’ or ‘or’ and group the conditions with parentheses.
* The ‘when’ keyword also supports a list and when using a list, all of the conditions must be true.
* ‘When’ can be used for ‘register’ elements.

Example for when with list: (multiple conditions)

tasks:

- name: tasks for when with list

yum:

name: httpd

state: installed

When: # all the conditions must be true.

- ansible\_distribution == 'RedHat'

- ansible\_memfree\_mb < 400 # here 400 in integer.

Example for when\_with\_loop:

tasks:

- name: play for demonstration of when with loop

package:

name: "{{ package }}"

state: latest

loop: "{{ ansible\_mounts }}"

when: item.mount == "/boot" and item.block\_size == 4096

Example when\_with\_and\_or:

tasks:

- name: removing the httpd

yum:

name: httpd

state: removed

when: >

( ansible\_memfree\_mb < 400 and ansible\_distribution == 'RedHat' ) or

( ansible\_distribution == 'CentOS' and ansible\_memfree\_mb < 400)

Example when\_with\_register:

tasks:

- name: tasks for when with register

user:

name: "{{ username }}"

register: reg\_when

- name: debugging the val

debug:

var: reg\_when['group']

- name: condition

debug:

msg: 'condition is successful.'

when: reg\_when['group'] == 1004

Example to find sub-string:

tasks:

- shell: cat /etc/passwd

register: passwd\_contents

- debug:

var: passwd\_contents

- debug:

msg: echo 'paswd contains user {{ username }}'

when: passwd\_contents.stdout.find(username) != -1

Note: In Python, the find method for strings is used to search for a substring within a string. It returns the lowest index of the substring if it is found. If the substring is not found, find returns -1.

!= -1: This checks if the find method did not return -1. If find returns -1, it means the username was not found in the string.

**Understanding Handlers:**

Handlers: Handlers are the tasks that only run if triggered by a task that has changed something.

* In order to run the handler, a ‘notify’ statement is used to notify the handler. ((example for these is in ‘handler.yml’).
* Handlers typically are used to restart services or reboot hosts.
* If **one of the next tasks** in the play fails, the handler will not run(it’s a default behavior). However this may be overwritten using ‘force\_handlers = yes’. (example for these is in ‘force\_handlers.yml’).
* One task may trigger more than one handler.
* Handlers are executed at the end of the play.
* To execute the handler **immediately after this point** use, ‘meta’ module. (example for these is in ‘meta\_handlers.yml’)

**Blocks**: can be used to implement if-then-else like statements.

Block is a logical group of tasks. (can be used for handling task errors (block, rescue and always.)

Used to control how many tasks are executed and can also be used in error condition handling:

i) use **block** to define the main tasks to run. (list of tasks is defined in a block).

ii) use **rescue** to define tasks that run if tasks defined in the block fails.

iii) use **always** to define tasks that will run, regardless of the success or failure of the block and rescue tasks.

Note: ‘items’ cannot be used on blocks.

Note:

Block-Level Condition:

The when condition at the block level (when: ansible\_distribution == 'CentOS') ensures the entire block only runs on CentOS systems.

Task-Level Conditions:

Individual tasks within the block can have their own when conditions. For example, when: ansible\_distribution == 'RedHat' for the yum task ensures this task only runs on RedHat systems.

**Managing Task Failure:**

Ansible looks at the exit status of a task to determine whether it has failed.

When any task fails, ansible aborts the rest of the play on that host and continues with the next host.

Different solutions can be used to change that behavior.

* Use ‘ignore\_errors’ in a task or play to ignore failures. (Can I use ignore errors in play header?
* No, the ignore\_errors directive cannot be used at the playbook or play header level in Ansible. It is only applicable at the task level.)
* Use ‘force\_handlers’ to force a handler that has been triggered to run, even if (another) task fails.

Note: Notice that if ‘ignore\_errors: yes’ and ‘force\_handlers: no’ both have been set, the handlers will run after failing tasks.

Ex: tasks:

- name: first play

command: echo hello world

ignore\_errors: yes

register: command\_result

failed\_when: "'world' in command.stdout"

- name: print the output

debug:

var: command\_result

**Defining Failure States:**

As ansible only looks at the exit status of a failed task, it may think a task was successful where that is not the case.

To be more specific, use ‘failed\_when’ to specify what to look for in command output to recognize a failure.

Note:

* The ‘failed\_when’ keyword can be used **in a task** to identify when a task has failed.(failed\_when directive in Ansible allows you to define custom failure conditions for a task.)
* The ‘fail’ module can be used to print a message that informs why a task has failed.

**Fail Module:**

To use ‘failed\_when’ or ‘fail’, the result of the command must be registered, and the registered variable output must be analyzed.

When using the fail module, the failing task must have ‘ignore\_errors’ set to yes.

Ex:

hosts: ansible1

ignore\_errors: yes

tasks:

- name: run a script

command: echo hello world

register: command\_result

- name: report a failure

fail:

msg: failed the command. # intentionally, failing the task.

when: "'world' in command\_result.stdout"

- name: see if we get here

debug:

msg: successful.

**Understanding Changed:**

Idempotent modules can make a difference between “changed” and “no-change required”.

Non-Idempotent modules like ‘command’ and ‘shell’ cannot do that, and only work with an exit status of 0 or 1.

(when you use command or shell module they’ll execute commands and ansible thinks it has changed (because module has run successfully)).

Because of this, a non-idempotent module may falsely report changes, when really no change has happened.

Managing the changed status may be important, as handlers trigger on the changed status.

Using ‘changed\_when’ is common in two cases:

* To allow handlers to run when a change would not normally trigger.
* To disable commands that run successfully to report a changed status.

Ex:

tasks:

- name: check local time.

command: date

register: command\_result

changed\_when: false

- name: print local time

debug:

var: command\_result.stdout

**Including and Importing Files:**

If playbooks grow larger, it is common to use modularity by using includes and imports.

Includes and imports can happen for roles, playbooks as well as tasks.

An **include** is a dynamic process, ansible process the contents of the included files at the moment that this include is reached. (keyword: ‘include\_tasks’ and its a module)

An **import** is a static process, ansible preprocesses the imported file contents before the actual play begins. (keyword: ‘import\_tasks’ and it’s a module.)

Note: You cannot trigger a handler in an imported task file from the main task file.

Note: store task files in a dedicated directory to make management easier.

To import a play from a import file, use ‘import\_playbook’

Note: imported play shouldn’t be a part of another play.

**LEsson - 8 (Mangaing files)**

**Modifying files:**

Modules used in ansible:

* **File** module in Ansible is a versatile tool for managing files and directories on remote hosts. It can be used to create, delete, modify permissions, and set various attributes of files and directories.
* **Copy** module to copy a file from source to managed node.
* **Synchronization** module for synchronizing the file to managed nodes but requires the **rsync** to be installed on all the hosts.
* **Fetch** module is used to fetch(pull) a file from a managed node to control the host.
* **Replace** module is used to replace a text in a file
* **Lineinfile** module is used to add a single line in a text file.
* **Blockinfile** module is used to add multiple lines in a text file.

Example for file module:

tasks:

- name: create a file

file:

path: /tmp/new\_file

state: touch

mode: 0770

Example for Blockinfile:

blockinfile:

path: "{{ file\_path }}"

block: |

This is the first line

and sennd line

Example for copy:

- name: copy the file content

copy:

src: /etc/hosts

dest: /tmp/

Example for fetch:

- name: tasks for fetching the file

fetch:

src: /tmp/hosts

dest: /tmp/\_host

flat: yes # directly uses the file name. From ‘dest’ and check the output, if ‘flat’ is not mentioned.

**Templates:** Templates are the sample configuration files that are combined with environmental variables to produce site specific configuration files.

Template has 2 components: 1) Template file(which is in format of jinja2 (.j2) and

2) Playbook that has a template module.

About **Jinja 2 format** file: It is used to modify the file with advanced features and can also be used to create dynamic and flexible configuration files.

NOte: for simple modifications use ‘lineinfile or blockinfile modules’.

Note: have to mention a default value in ‘ansible.cfg’.

Ansible\_managed = {file} is modified by Ansible date is %d-%m-%Y user id is {uid}

Referring to variables ‘file’ is jinja2 specific and ‘uid’ is from linux.

Example of jinja2 template:

{{ ansible\_managed }}

anonymous\_enable={{ anonymous\_enable }}

local\_enable={{ local\_enable }}

write\_enable={{ write\_enable }}

anon\_upload\_enable={{ anon\_upload\_enable }}

dirmessage\_enable=YES

xferlog\_enable=YES

connect\_from port\_20=YES

pam\_service\_name=vsftpd

userlist\_enable=YES

#my IP address={{ ansible\_facts['default\_ipv4']['address'] }}

—-----Playbook that access the variables:

name: playf or vsftpd

hosts: ansible1

vars:

anonymous\_enable: yes

local\_enable: yes

write\_enable: yes

anon\_upload\_enable: yes

tasks:

- name: install vsfttpd

yum:

name: vsftpd

state: installed

- name:

template:

src: vsftpd.j2 # source of jinja2 template.

dest: /etc/vsftpd/vsftpd.conf

Note: Practice jinja2-template. For RHCE. In ‘docs.ansible.com’ look for frequently asked questions, ‘For’ loop and ‘template’ examples.

Example for template with for loop:

**#Accessing the Jinja2 template:**

name: tasks for jinja2

template:

src: template/for\_jinja\_temp.j2

dest: /tmp/jinja2 # destination of remote host or managed node.

**# template file with for loop:**

{% for host in groups['db\_server'] %}

{{ hostvars[host]['ansible\_default\_ipv4']['address'] }}

{% endfor %}

**Managing SeLinux file context:**

File module can be used to manage sellinux but it won’t write to the selinux policy(it will write into the file system).

‘Sefcontext’ module will provide context to policy. And then run the ‘restorecon’ command to write into the file system.

Example:

tasks:

- name: tasks for

sefcontext:

target: /tmp/new\_file

setype: public\_content\_rw\_t

state: present

notify:

- run restorecon

handlers:

- name: run restorecon

command: 'restorecon -v /tmp/new\_file'

Note: (for exam) there are magic variables that can be found in ‘docs.ansible.com’, under the playbooks section and search for magic variables.

Example:

tasks:

- name: set hosts

command: hostnamectl set-hostname {{ inventory\_hostname }}

# {{ inventory\_hostname }} is a magic variable.

**Roles:** Roles are the community provider resources that makes working with ansible easier.

It provides a standardized solution for common tasks.

(Roles are about reusable code).

Note: when creating custom roles, it should only consist of tasks.

**USing ansible-galaxy to get roles:**

Getting roles from external sources:

* Through ansible galaxy, either as roles or as a part of a content collection.
* As a ‘tar’ file.
* From RPM packages.

Installing roles:

command : ‘Ansible-galaxy role install role\_name’

Get the role name from ‘galaxy.ansible.com’.

While installing roles, the ‘roles\_path’ setting is used.

By default, ‘roles\_path’ will use roles in the following order of precedence:

1. A roles directory in the current project directory.
2. The ‘~/.ansible/roles’ directory.
3. /etc/ansible/roles (note: it’s for generic and could be for custom roles.)
4. /usr/share/ansible/roles. (Note: if you are installing RPM packages that contain roles, it’s a default location).

Note: All the above points must be mentioned in ‘ansible.cfg’

Optionally, use ‘ansible-galaxy role install -p path\_of\_role\_file’ to install in a specific path.

**Searching a role with in CLI:**

ansible-galaxy role search 'docker' --author sebastian13 --platform EL

Here, ‘docker’ is role name,

Author name is ‘sebastian13’ and it must support the platform of ‘Enterprise Linux(EL)’.

**Roles in Playbooks:**

**Different ways to define roles:**

* Roles are defined in the play header. (and these roles will be executed before the tasks in the play.)
* Using the ‘import\_role’ or ‘include\_role’ module in a task.
* ‘Pre\_tasks’ to trigger tasks to run before the roles.
* ‘Post\_tasks’ to force tasks to run after the roles.

Note: (Best practices): if roles should be executed after some tasks, define the tasks and use ‘import\_role’ where needed.

**Include\_role v/s Import\_role:**

* Inlcude\_role dynamically includes the role when it is referred to in a task. (could be used with when statement)
* Import\_role is processed when the playbook is parsed(render).
* While using ‘import\_role’ the role handlers and variables are exposed to all tasks in the play, even if the role has not run yet.

NOte: in most cases, using ‘include\_role’ is better than using import\_role.

**Variables in Roles:**

1. Roles are often pre-configured with standard variables:

* Variables in the ‘default’ directory in the role, (this can be change in plays)
* Variables in the ‘vars’ directory in the role are used for internal purposes in the role(these are not intended to change in the playbook).

Note: ‘default and vars directories’ are present in roles.

1. Site-specific information should be set through playbook variables, which will be picked up by the role.

Note: Don’t define local variables or vault encrypted variables in the role, they should always be defined locally in the playbook.

**Define variables are Role Parameters:**

* Variables can be set as a role parameter while calling the role from the playbook.
* This is useful if the same role is used multiple times, with different values for the same variables.

Ex:

* Hosts: webservers

Roles:

* Role: role\_name

Message: hello

- ROle: role\_name

Message: bye.

**Role variable precedence:**

* Role variables in the **vars** directory are not supposed to be overwritten, but will be overwritten by facts, registered variables, and variables loaded with include\_vars.(said by sander van vagt)
* Role variables in the **default** directory are overwritten by any other variable definition.
* If a variable is declared as a role parameter, it has the highest precedence.

**Write the Custom Roles:**

Create a directory for a role (role\_name). It must be present in roles (location: ~/.ansible/roles) and mention the location in ‘ansible.cfg’ (in ‘roles\_path’).

Cmd: ansible-galaxy role init role\_name # it initializes the new role with the directory structure of a role.

→ Define whatever the parameters you want in:

In meta(meta information): it could have ansible version info like 2.9

In tasks ( where the actual tasks is present)

Templates: define the template for jinja2.

Till here your role is ready.

# Define a playbook that access the role

* Name: play for accessing the role

Pre\_tasks:

* Name: to do a task before the role.

Debug:

Msg: print the message before the execution of the role.

Roles:

- role: role\_name.

**Note**: when creating custom roles, it should only consist of tasks. (No play headers or anything).

**RHEL system roles:** They are provided as an easy interface to manage specific parameters between different RHEL versions. (parameters could be for Networking or time management)

(to manage Networking or time management there were different commands based on the versions).

**Installing RHEL system Roles:**

Note: (for Exam) install rhel-system-roles using ‘dnf-install rhel-system-roles’.

This installs them to **/usr/share/ansible/roles** and /**usr/share/ansible/collections**, which ensures easy access from ansible-navigator as well as ansible-playbook utility and also provides useful examples in ‘/usr/share/doc’.

Note: for time synchronization, get the example from ‘/usr/share/doc’.

Example for NTP server:

# This playbook is going to sync ansible1 with ntp-server.

- name: Example with single pool

hosts: ansible1

vars:

timesync\_ntp\_servers:

- hostname: 2.pool.ntp.org

pool: true

iburst: true

roles:

- rhel-system-roles.timesync

Example for SELinux:

Example of selinux could be found in ‘/usr/share/rhel-system-role/selinux/example-selinux.yml’

Change the parameters as per the requirements.

**TroubleShooting without logs:**

* Ansible ad-hoc command and ansible-playbook command gives the stdout on the screen.
* When using ‘ansible-navigator’ use ‘-m stdout’, which will give the output stdout.
* When executing a playbook, ‘PLAY RECAP’ section is shown below.
* Use the verbosity. Example:’ansible-playbook -vvv playBook\_name’

**Logs:**

* Ansible by **default** doesn’t store the logs. To store logs, in ‘ansible.cfg’ mention ‘log\_path=/path/of/file’.

NOte: don’t use ‘/var/log’, ‘cause it requires privileges.

* When you use ‘ansible-navigator’ it automatically stores artifacts(relevant information of the playbook) in the current directory.

(for **not storing artifacts**, create a ‘ansible-navigator.yml’ and mention):

---

ansible-navigator:

Playbook-artifact:

enable: false

**Using the debug module:**

This module is used to analyze the variables and print statements. You can use the ‘verbosity’

Example:

- name: tasks for log

debug:

var: "{{ ansible\_hostname }}"

verbosity: 1

**Checking playbooks for issues:**

* Use ‘--syntax-check’ to check the syntax.
* To avoid issues use the best practices:

1. Use consistency in whatever you do.
2. Make sure every task and play has a name that describes what the play is doing/
3. Use consistent indentation.
4. In case of doubt, use the ‘debug’ module to check for variable values.
5. Keep it simple.
6. Always use the most specific ansible solution.
7. Avoid non-idempotent modules.

NOte: ‘ansible-list’ checks the playbook against the best practices.

**Check Mode:** it would check what would happen when running the playbook but it won’t change anything.

Command: Ansible-playbook --check playbook\_name

If you don’t want a specific task to be checked, mention: ‘check\_mode: no’ in that task.

* Modules in the playbook must support check mode.(few modules doesn’t support)
* Check mode doesn’t work well in conditionals.

**Modules that can help for Troubleshooting Ansible:**

Uri: check the content that is returned from a specific URL.

Script: runs a script from the control node on the managed hosts.

Stat: checks the status of a file or directory. (use it to register a variable and next test to determine if a file exists.)

Assert: It’s a module and can be used to print a message on success or failure.

Advantage of Assert: it allows you to write multiple conditions.

Example: tasks:

- name: tasks for

Package\_facts: # its module, Return information about installed packages as facts.

manager: auto

- name: print the value

debug:

msg: "version {{ ansible\_facts.packages['bash'][0]['version'] }}" # bash is dict and [0] is list item, and the item consists of dict.

when: "'bash' in ansible\_facts.packages"

**Example to check a file (using stat module):**

tasks:

- name: tasks for the stat module

stat:

path: /etc/hosts

register: reg

- name: task for print the reg

debug:

var: reg.stat

- name: task for check the file

debug:

msg: 'file does exists'

when: reg.stat.exists

**Example for assert module:**

vars\_prompt:

- name: value

prompt: enter the value # it stores the string.

private: no

tasks:

- name: tasks for module

assert:

that:

- ( value | int ) <=100 # convert value into ‘int’

- ( value | int ) > 10 # convert value into ‘int’

fail\_msg: 'condition fails'

success\_msg: 'successfullly value: {{ value }}'

**Connection Issues in Ansible:**

1)Issues can be of physical connection or Issues running tasks as the target user.

2) Authentication Issues:

i) Confirm the remote\_user settings(in ansible.cfg at control node) and existence of remote users on the managed hosts.

ii) Confirm host key setup ( do ssh login to managed hosts to just verify whether it’s logging in or not).

iii) verify ‘become=true’ in **playbook** and ‘become\_user=username’ in **ansible.cfg.** ( for privilege escalation)

iv) check that **sudo** is configured correctly. ( sudo is not required for logging in. A few commands require the root privileges. Check ‘**visudo’**)

3) Connecting to Managed hosts:

When a host is available at different IP addresses / names, you can use ‘**ansible\_host’** in inventory to specify how to connect. (note it’s a persistent way to connect). Examples could be found in the inventory file (/etc/ansible/hosts).

4) Ad-hoc commands to test the connectivity:

Use of a ping module, ‘ansible managed\_node\_host -m ping’ # checks whether the managed nodes are ping able or not.

**Assignment to check NIC exist or not:**

**-** name: assret

assert:

that:

# - res['ansible\_eth0']['device'] == "eth0"

- ansible\_eth0 is defined # is it defined or not

fail\_msg: 'failded'

success\_msg: 'exist'

**Managing Software packages:**

Different modules are available for managing software packages:

* ‘Dnf’ is used in RHEL 9.
* ‘Yum’ is used for older versions of RHEL.
* ‘Apt’ is used to manage packages on Ubuntu and other distributions.
* Module name “package” is a generic package downloader for all distributions. (however, it might not download some additional features for distributions.)

NOte: Realize that package names are not always the same between different distributions.

Managing package groups:

‘Dnf group list’ to show a list of available groups.

To install a package group, put a ‘@’ in front of the group name:

* Name: name

Dnf:

Name: ‘@Virtualization Host’

State: present

# installing module with specific verison.

* Name: install php module

Dnf:

Name: ‘php:7.3/minimal’

State: present

Note: ‘setup’ module doesn’t gather the facts. To gather the facts use the ‘package\_facts’ module.

Example:

- name: tasks for

Package\_facts: # its module, Return information about installed packages as facts.

manager: auto

- name: print the value

debug:

msg: "version {{ ansible\_facts.packages['bash'][0]['version'] }}" # bash is dict and [0] is list item, and the item consists of dict.

when: "'bash' in ansible\_facts.packages"

**Managing Repository and Repository access:**

* To access repositories, the ‘yum\_repository’ module is used.

(It will create repo file in ‘/etc/yum.repo.d/’)

NOte: ‘gpgcheck’ is good for internet repo.

**Note: check the example in documentation.**

**Assignment: (Managing Repositories)**

1. Setup a repository(local repo) on control. This repository should offer multiple packages, including the nmap package.
2. Provide a package list using variables.
3. Configure ansible1 and ansible2 to use the repository that is provided through this repository.
4. Install the nmap package from this repository.

**Ans:**

- name: play for setup repo (control server)

hosts: ansible1

tasks:

- name: task for installing ftp server and ‘createrepo\_c’ command.

yum:

name:

- vsftpd

- createrepo\_c

state: latest

- name: start the vsftpd

service:

name: vsftpd

state: started

enabled: yes

- name: play for setting up repo

hosts: ansible1

tasks:

- name: make the directory

file:

path: /var/ftp/repo

state: directory # creating a directory

- name: download packages

yum:

name: nmap

download\_only: yes # just download files, do not install it.

download\_dir: /var/ftp/repo # downloading files to a specific location.

- name: creating repo

command:

cmd: createrepo /var/ftp/repo # it will create a repo.

#### Client machines

- name: play for the ansible2

hosts: ansible2

vars:

- mypackage: nmap

tasks:

- name: get the package facts

package\_facts:

manager: auto

- name: package facts

debug:

var: ansible\_facts.packages

- name: connect to example repo

yum\_repository:

name: lesson11

description: YUM repo

file: lesson11.repo

baseurl: ftp://ansible2/repo/

gpgcheck: no

- name: install package

yum:

name: "{{ mypackage }}"

state: present

- name: updating the package facts

package\_facts:

manager: auto

- name: facts

debug:

var: ansible\_facts.packages[mypackage]

when: mypackage in ansible\_facts.packages

**Lesson 12 User Management:**

Modules that are used for user management:

* User: manager core user properties.
* Group: create and manage groups.
* Known\_hosts: this module copies host keys from managed hosts.(This ensures that users are not prompted to verify the remote host SSH key fingerprint before connecting to it.) (updates the ‘/etc/ssh/ssh\_known\_hosts’ file with the host key of a managed host.)
* Authorized\_keys: module can be used to copy a control host user public key to the corresponding user account(target user) on a managed host. (manages authorized\_keys for user accounts on managed hosts.)

(to use it, the public key must be in a public location, where it is readable: if it is in a hidden directory in the user home directory it cannot be used.)

**Note**: This module is useful for ansible management users,that already exists on the control host and to which you need to set up the public key on the managed node.

**Important points**:

* before creating a user, create a group and then create a user.
* If a user needs to be a member of additional (secondary) groups, ensure these groups exist before using the user module.
* While adding users to secondary groups, use the ‘append:true’ option.

(or else it’s going to override the existing group.)

Example:

- name: create remote management user.

hosts: ansible1

become: true

tasks:

- name: create user ansible ( on remote machine).

user:

name: ansible

- name: give sudo privileges.

lineinfile:

path: /etc/sudoers.d/ansible

state: present

mode: 0440

line: 'ansible ALL=(ALL) NOPASSWD: ALL'

validate: /usr/sbin/visudo -cf %s

- debug:

msg: 'The remote management user is now created and has sudo privileges.'

- name: manage user keys

hosts: localhost

become: true

tasks:

- name: create a directory to store the file that authorized keys is going to distribute.

file:

name: ansiblekey

state: directory

- name: copy the local user ansible ssh key to this directory

shell: "cat /home/ec2-user/.ssh/id\_rsa.pub > ansiblekey/id\_rsa.pub" # copy module is not efficient here.

- debug:

msg: 'the local management user key is now in a place where it can be used.'

- name: create another remote user with a key.

hosts: ansible1 # Imagine it's a remote machine.

become: true

tasks:

- name: copy the managment user authorized key to the management host

authorized\_key:

user: ansible

key: "{{ lookup('file', './ansiblekey/id\_rsa.pub') }}" # reading file from control node

- name: create a remote user with an ssh key pair

user:

name: johncena

generate\_ssh\_key: true

ssh\_key\_bits: 2048

ssh\_key\_file: .ssh/ansiblekey\_rsa

**Managing Encrypted password:**

* Create a ‘vars’ directory and define variables(user and mypassword) in a text file or yml file and encrypt it with the ansible vault.
* In the playbook create a user with user module and then run shell module as

‘echo {{ mypassword }} | passwd --stdin {{ user }}

**Managing Sudo Privileges:**

* Ansible doesn’t provide a ‘sudo’ module.
* Use generic modules instead:

1. ‘Lineinfile’: can be used to manage the ‘/etc/sudoers.d/whatever’ file
2. ‘Template’: can be used with a jinja2 template to generate this file.

Note: use ‘validate: /usr/sbin/visudo -cf %s’ to validate your modification

Example:

# default file which consists of variables.

sudo\_groups:

- name: developers

groupid: 5000

sudo: false

- name: admins

groupid: 5001

sudo: true

- name: dbas

groupid: 5002

sudo: false

# jinja2 file

{% for item in sudo\_groups %} # looping over the items

{% if item.sudo %} # checking each item is true.

%{{item.name}} ALL=(ALL) NOPASSWD:(ALL) # line to be added.

{% endif %}

{% endfor %}

# main file ( setup\_sudoers.yml)

vars\_files:

- host\_vars/defaults

tasks:

- name: adding groups to sudoers

template:

src: temp.j2

dest: /etc/sudoers.d/sudo\_groups

**Lesson 13:**

**Managing services and targets:**

Service: it's a module that can be used to manage services.

Reboot: can be used to reboot the machine.

Systemd: controls systemd units on remote hosts.

Note: there is no module for the targets.

Ex:

vars:

- target: 'multi-user.target'

tasks:

- name: task for default

command:

cmd: 'systemctl get-default'

register: res

changed\_when: false # changed\_when: used in the task to make it idempotent.

# command module will give you ‘changed’ when it gets executed. To override this, we’ll use ‘changed\_when’.

- name: set default target

command:

cmd: 'systemctl set-default {{ target }}'

when: target not in res['stdout']

notify: reboot\_server

handlers:

- name: reboot\_server

reboot:

test\_command: uptime

reboot\_timeout: 100

**Scheduling Process:**

Scheduling a cron job with ansible:

Modules: ‘at’(only works once) and ‘cron’ modules are used to create a cron job.

Example:

tasks:

- name: creating a cron.

cron:

name: "create cron"

minute: "\*/2"

hour: 16-18

job: "echo 'new line' >> /tmp/cron\_created"

user: ec2-user

cron\_file: newcron

state: present

**Lesson 14: Managing Storage:**

Modules used are:

* Ansible.posix.mount # is used to mount existing filesystems.
* Community.general.parted # is used to manage partitions.
* Community.general.lvg # manages **volume groups.**
* community.general .lvol # manages **logical volumes**
* Community.general.filesystem # can be used to create filesystems on the new devices.

NOte: if community.general content collection is not working, try using ‘redhat.rhel\_system\_roles.storage’ role. Or you can use the ‘command’ module.

Sander says use either of it. Because the command module is not idempotent. And the ‘storage’ role’ is inconvenient.

Example: Creating a partition on 1GB:

tasks:

- name: task for creating a partition.

parted:

device: /dev/xvdc

number: 1

state: present

part\_end: 1GiB

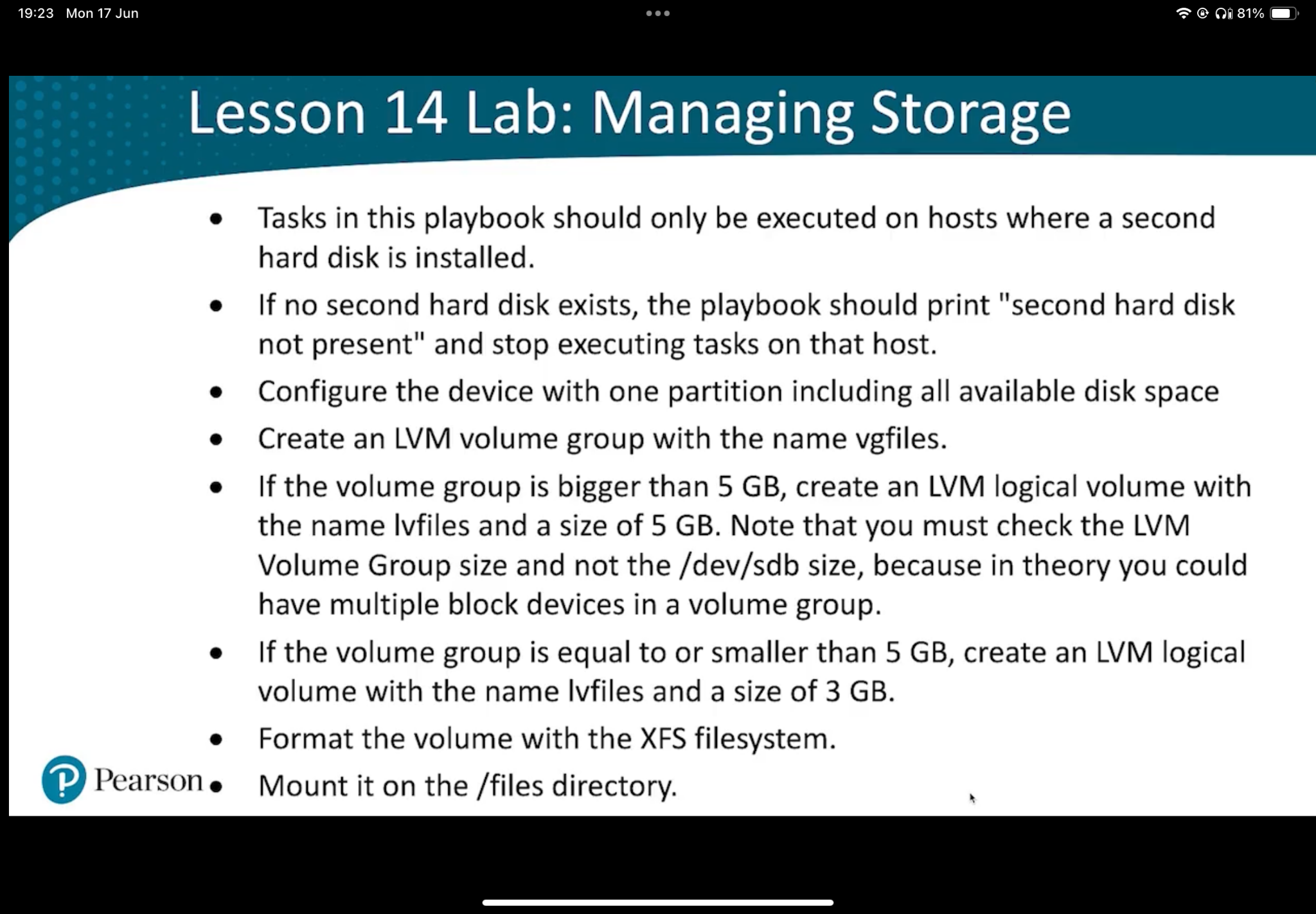
when: ansible\_devices['xvdc'] is defined # checks whether the device is present.

**Important Note:**

**Developing Advanced Playbooks:**

* First, set up a generic framework(basic structure of the playbook) that shows how you are going to take care of any conditional task execution. In this phase don’t use specific modules but put as much as you can in the ‘debug’ module and focus on structure.
* Next, work out the conditional and facts you may want to check. Still no details about specific modules but check the facts using the debug module.
* Finally, work out module specific and produce the working solution. In this phase, consider using tags to allow you to test specific parts only.

**Assignment 14lab:**

****

**Ans:**

tasks:

- name: checking whether it exist or not.

fail:

msg: device does not exist

when: ansible\_devices['xvdc'] is not defined

- name: configuref the device with 1 partition.

parted:

device: /dev/xvdc

number: 2

state: present

part\_start: 1GiB

part\_end: 4GiB

#var: ansible\_devices['xvdc']['size']

- name: create LVM volume group.

lvg:

vg: vgfiles

pvs: /dev/xvdc2

- name: creating variable to check

set\_fact:

vgsize: "{{ ansible\_lvm['vgs']['vgfiles']['size\_g'] | int }}"

- name: creating logical volume.

lvol:

vg: vgfiles

lv: lvfiles

size: 5GiB

when: vgsize | int > 5

- name: creating lvm

lvol:

vg: vgfiles

lv: lvfiles

size: 2.5g

when: vgsize | int <= 5

- name: creating directory with file

file:

path: /files

state: directory

mode: '0755'

- name: create filesystem

filesystem:

fstype: xfs

dev: /dev/vgfiles/lvfiles

- name: formating volume and mount it on /files

mount:

path: /files

src: /dev/vgfiles/lvfiles

fstype: xfs

state: mounted

**Note:**  There are few examples that are present in ‘/usr/share/doc/rhel-system-roles’. Perhaps it would help in the exam.

**LEsson - 15 ( Network Management):**

* The ‘redhat.rhel\_system\_roles.network’ system role allows for the configuration of network related settings.
* To configure the role, the ‘network\_provider’ and ‘network\_connections’ variables must be set:

1. ‘Network\_provider’: should be set to ‘nm’ on RHEL 7 and later versions.
2. ‘Network\_connection’: defines the network connection and its properties.

**Modules That are useful to manage Networking and Facts:**

* Firewalld: this module creates and manges the firewalls.
* Hostname: is used to set the hostname of the system.
* Facts: ansible\_facts[‘interfaces’] # shows the interfaces of the managed node.

Note: more examples could be found in ‘**/usr/share/doc/rhel-system-roles/specifiC\_role/READ.md’**

**Task:**

**Creating Hash password for user:**

tasks:

- name: creating hash passwords

set\_fact:

hashed\_passwords: "{{ hashed\_passwords | default({ }) | combine({ item.name: (item.password | password\_hash('sha256'))}) }}"

loop: "{{ users }}" # loop over the list.

**Explanation:**

Hashed\_passwords = is variable

default({ }) is a filter, used if there is empty value it will store empty value.

Combine is a filter, used to combine key-value pairs. ( item.name is user name, item.password is user\_password and password\_hash(‘sha256’) is a filter to create a hash password.)

Note:

* when you create bash script, there should be execute permissions on it.
* When you edit the config file make sure to restart the service.
* In order to verify any stuff in ansible use debug, stat module and command module, store it in register.
* For templates, when using loop(for) and condition(if) in templates mention ‘%’ at start and end.
* For devices, ansible fact variable would be ‘ansible\_device’
* For lvms or vgs, the fact variable would be ansible\_lvm ( more important have to download the ‘lvm’ module.(present in ‘community.general’ collections.)
* If the ansible.cfg is using the default config file. To change it, edit ‘.bashrc’ and add a line ‘export ANSIBLE\_CONFIG=config\_file\_path’. Then execute it with ‘source .bashrc’.
* For extracting hosts ip-addr, fqdn. Hostname in template file use:

{% for host in groups['all'] %}

{{ hostvars[host]['ansible\_facts']['default\_ipv4']['address'] }} {{ hostvars[host]['ansible\_facts']['fqdn'] }} {{ hostvars[host]['ansible\_facts']['hostname'] }}

{% endfor %}

Explanation: when using ‘ansible\_facts’ in file don’t mention ‘ansible\_’ for keys.

‘Cause we are mentioning ‘ansible\_facts’, it’s more than enough.

Note: you can use ‘default(‘NOne’)’ function in template as well.

A task to do: (temp.j2)

-- inventory hostname: {{ ansible\_facts['hostname'] }}

-- Total memory in MB: {{ ansible\_memtotal\_mb }}

-- BIOS version: {{ ansible\_bios\_version }}

-- Size of disk device vda: {{ ansible\_devices['xvdf']|default('None') }}

-- Size of disk device vdb: {{ ansible\_devices['xvde']|default('None') }}

Access this in (.yml): by using template module.

Note: when using condition, and comparing 2 values like

when: ansible\_devices['xvdd']['size'] > '4.0'

Use string on both the sides.