RHCE Study Guide

**Understanding Configuration Management**

* Devops manages the application lifecycle: (1) Coding (2) Building (3) Testing (4) Packaging (5) Releasing (6) Configuring (7) Monitoring
* Infrastructure as Code (IaC) states the desired state of managed IT infrastructure to ensure compliance . The code is managed in a concurrent version system (CVS).

**Ansible Essential Components**

* Controller node: runs Ansible software (must be Linux)
* Managed node: connected to via the controller and kept in the controller's inventory
* Managed nodes are configured in Ansible playbooks written in YAML to run plays
* Modules: used to implement the tasks in plays via code (can be enhanced with plugins)
* Playbooks are idempotent, meaning they produce the same result every time. They should also be self-contained (not dependent on other playbooks)
* Ansible Engine=CLI version used for exam. Ansible Tower=web-based GUI

**The Ansible Way**

* Keep it simple and make it readable
* Use the declarative approach, meaning change the current state into the desired state
* Use specific solutions/modules, not just commands on managed nodes

**Ansible Use Cases**

* Configuration management: manages the configuration files, install software, create users, and perform similar tasks for a desired state. The primary use case
* Provisioning: deploy and install systems
* CI/CD: helpful for continuous delivery and deployment

**Setting Up an Ansible Managed Environment**

**Ansible Projects**

* Project directories can/should be used as they are a self-contained environment with everything needed to work on a specific project such as: playbooks, inventories, variables, .cfg files, etc.

**Configuring a Static Inventory**

* Used to:
* Identify hosts
* List hosts
* Group hosts

\*TIP\* use /etc/ansible/hosts for an example

* Projects typically have dedicated host files whereas smaller environments may use /etc/ansible/hosts
* Static inventories can be a list of hosts and/or IPs. They can also be ranges, i.e. server[1:6].example.com

**Inventory Host Group**s

* Hosts can be placed in one or more groups

Example:

[web]

web1

web2

[db]

db1

db2

* Groups can be functional, regional, or staging (implementation phase)
* Host files can have hosts that are just listed ungrouped in addition to grouped ones
* Besides defined host groups, there are a few implicit hosts and groups in Ansible:
* localhost
* all (all hosts in an inventory)
* ungrouped (any host not in a group)

**Using Inventory in Commands**

* ansible <specify hosts/group> -i inventory --list-hosts (list hosts in specified group for specified inventory file)
* ansible-inventory -i inventory --graph (shows hosts in inventory file in graphical view)
* The inventory can be specified via relative or absolute path

**Working with Dynamic Inventory**

* Small environments typically use manually defined inventory but larger ones can use scripts for discovering inventory hosts (usually use .ini extension)
* Community versions of dynamic scripts can be found
* You can write your own but they must contain a --list and --host option
* More than one inventory can be used as long as they are put into the directory as an inventory file and have the executable permission set

**Managing Settings in ansible.cfg**

* ansible.cfg file contains settings for reaching out to managed hosts

\*TIP\* the /etc/ansible/ansible.cfg file is a good resource

* Section headers are between [ ] and parameters are set in key=value pairs
* Common sections are [defaults] for general info and [privilege\_escalation] for admin user privileges

**Common ansible.cfg Settings**

remote\_user = name of user account to connect to on managed node

inventory = relative or absolute filename for inventory to use

become = boolean for whether privilege escalation is needed

become\_method = which local mechanism to use for privilege escalation

become\_user = name of user account for become\_method to run commands as

become\_ask\_pass = whether a password should be asked for when running privileged commands

host\_key\_checking = whether SSH host key should be checked

Example .cfg file

[defaults]

remote\_user = ansible

host\_key\_checking = false

inventory = inventory

[privilege\_escalation]

become = true

become\_method = sudo

become\_user = root

become\_ask\_pass = false

Using Ad Hoc Commands

* Ad hoc commands are commands that can be run without playbooks (good for setup tasks, quick tests and discoveries)
* Eg. ansible all -m user -a "name=lisa" (tests all hosts using the user module and the name argument to see if lisa exists)

Working with Modules

* Modules provide ansible's functionality
* Examples of modules include:
* command = run commands with no shell (no pipes or redirects, etc)
* shell = run commands using a shell
* raw = run commands directly on top of SSH without Python (good when python is not installed)
* copy = copy files or lines of texts to files
* yum = package management
* service = service management
* ping = ping

Browsing Available Modules

* The ansible-doc -l command will list all available modules
* The ansible-doc <module name> brings up the module's documentation
* ansible-doc is made up of:
* module name = name
* maintainer info = responsible party
* options = all arguments that can be used
* see also = related modules
* author = who wrote this module
* examples = sample code
* return values = expected results

\*TIP\* Use ansible-doc -s <module name> for a sample playbook for the module and utilize the examples section!

YAML & Playbook Elements

* Playbooks are written in YAML and saved with .yml or .yaml extension
* Composed of tasks
* YAML uses key=value pairs and always begin with --- at the top and uses - to specify embedded objects
* some modules can take more than one value for a key

Example:

yum:

name:

- vsftpd

- sshd

- httpd

state: latest

* Text strings can have single, double, or no quotes (your choice)
* Using the | symbol causes a string to be read normally while a > sign puts everything on one line
* Usually starts with a name, then hosts, then list of tasks with modules for each task
* Indentation is used for hierarchy
* Data elements at the same level in the hierarchy must have same indent
* Items that are children of another element get indented more (space bar twice)

\*TIP\* add the line autocmd FileType yaml setlocal ai ts=2 sw=2 et in ~/vimrc for auto-indentation

\*TIP\* the command ansible-playbook -C (stands for check) does a dry run that checks for syntax

Example playbook structure:

---

- name: install and start httpd <----name of play

hosts: all

tasks: <----start of task list

- name: install package <----name of task

yum: <----module

name: httpd <----argument

state: installed <----argument

- name: start httpd <----name of task

service: <----module

name: httpd <----argument

state: started <----argument

Running Playbooks

* command ansible-playbook <name>.yml to run a playbook
* Output will show the play name, tasks, whether it was successful, then a recap

Managing Multiplay Playbooks

* if a playbook holds more than one play, as long as it is indented properly you can specify them all in one yaml file

Example

- name: play 1

hosts: all

tasks:

- name: task 1

service:

name: httpd

state: latest

- name: play 2

hosts: ansible2

tasks:

- name: task 2

command: ping 127.0.0.1

Increasing Output Verbosity

* -v = more output including task results
* -vv = task results and task configuration, input/output (good for troubleshooting)
* -vvv = same as above plus info about connection to managed hosts
* -vvvv = same as above but adds info about plug-ins, users, and scripts

Understanding Variables in Ansible Playbooks

* Variables are labels that refer to data that can be dynamically changed (some variables are inherent and don't need defining such as ansible\_hostname)

Example:

- name: create user using a variable

hosts: all

var:

user: lisa <----variable defined in vars section of header

tasks:

- name: create the user {{ user }} on ansible\_facts['hostname']

user:

name: "{{ user }}" <----calling the variable

* Variable precedence: (1) variables passed on the CLI (2) variables defined in the playbook (3) inventory files

Working with Ansible Facts

* Three types of variables in Ansible:
* fact: variable that contain values describing specific system properties. Gathered on remote systems and can be used as variable. Can also be manually set and stored as a file on managed hosts.
* variable: variables defined by user
* magic variable: automatically set

Gathering Facts

* Ansible playbooks have implicit fact gathering tasks
* Task gathering results are stored in the ansible\_facts variable
* To address specific values in this dictionary, some common facts are:
* ansible\_facts['hostname'] = short hostname
* ansible\_facts['distribution'] = linux distro
* ansible\_facts['default\_ipv4']['address'] = main ipv4 address
* ansible\_facts['interfaces'] = list of network interfaces
* ansible\_facts['devices'] = list of attached storage devices
* ansible\_facts['distribution\_version'] = version of distro

\*TIP\* use ansible-doc -l | grep fact for fact gathering modules

Working with Custom Facts

* Custom facts are can be used to give arbitrary values to hosts
* Static files with containing custom facts use the INI or JSON format, have a .fact extension, and are stored on managed hosts in /etc/ansible/facts.d

Example:

[packages]

web\_package = httpd

ftp\_package = vsftpd

[services]

web\_service = httpd

ftp\_service = vsftpd

Working with Variables

* To define a variable (which are case sensitive and must start with a letter), use the key: value structure in vars section of header (seen in last example)
* If the variable is the first word of the value, you must use quotes, like name: "{{ web\_package }}"

Using Include Files

* Variables can be defined in the playbook vars section or in a file using the vars\_files: statement (can be a list or single value)

Example:

---

- name: using a variable include file

hosts: ansible1

vars\_files: vars/common <-----refers to a file in vars/common

vars/common file being referenced in the header looks like:

my\_package: nmap

my\_ftp\_service: vsftpd

my\_file\_service: smb

Managing Host and Group Variables

* Using host\_vars and group\_vars allows variables for specific hosts/groups
* To use them, a directory under the project must be made called host\_vars or group\_vars, eg group\_vars/ansible1 or group\_vars/webservers

Using Multivalued Arrays

* 2 types of arrays are (1) list and (2) dictionaries
* Each list item starts w/ a - and can be referenced by an index # (starts with 0) such as {{ users[1] }} for lisa

Example:

- linda:

username: linda

homedir: /home/linda

shell: /bin/bash

- lisa:

username: lisa

homedir: /home/linda

shell: /bin/bash

* Dictionaries are key:value pairs (same as arrays but no dashes)

Working with Magic Variables

* Auto-set by Ansible to reflect an Ansible internal state (cannot be used for anything else)
* host\_vars = contain all hosts in inventory + their variables
* groups = all groups in inventory
* group\_names = list groups this host is currently in
* inventory\_hostname = the inventory host name for current host
* inventory\_file = the name of the current inventory file that is used

Using Ansible Vault to Manage Sensitive Values

* Ansible vault is used to encrypt and decrypt sensitive data instead of storing it in plaintext playbooks. It works at:
* Sensitive data is stored as values in variables in separate variable files
* The variable file is encrypted using the ansible-vault command
* While accessing the variable file from a playbook, a password is entered to decrypt

Managing Encrypted Files

* When ansible-vault create secret.yml is used. A password request appears and opens the file using the default editor
* ansible-vault encrypt can be used to encrypt existing files and then utilized in a playbook after password authentication, ansible-decrypt does the opposite
* Ansible Vault commands:
* create = create new encrypted file
* encrypt = encrypt and existing file
* encrypt\_string = self-explained
* decrypt = self-explained
* rekey = change password on encrypted file
* view = show contents of encrypted file
* edit = edit an existing encrypted file

Using Vault in Playbooks

* ansible-playbook --vailt-id @ prompt = The option for ansible-playbook to prompt for a password on a vault-encrypted file
* To use vault encrypted files that all use the same password use the ansible-playbook --ask-vault-pass
* Use ansible-playbook --vault-password-file=secret to view vault password from a file

Managing Files with Sensitive Variables

* It is recommended to separate files with encrypted and unencrypted variables using group\_vars or special directories

Capturing Command Output Using Register

* The register parameter in a task can use command results as a variable

Example:

tasks:

- name: test register

shell: cat /etc/passwd

register: password\_contents

- debug:

var: "passwd\_contents"

* Register stores the command output in different areas (all shown in playbook results):
* cmd = command that was used
* rc = return code (0=success)
* sterr = error message generated (if any)
* stderr\_lines = error message line by line
* stdout = command output
* stdout\_lines = command output line by line

Loops and Items (Task Control)

* Some modules enable lists that need processing, for those that don't, a loop for iteration can be used

Example:

- name: install packages <-----yum does support lists

yum:

name:

- vsftpd

- httpd

- samba

state: latest

- name: start services <----service module does not support lists

service:

name: "{{ item }}" <----item is an internal system variable, doesn't need to be defined

state: started

enabled: yes

Loop:

- vsftpd

- httpd

- smb

Using Loops on Variables

* It is better to define variables in the header or in a file or host/groups

Example:

---

- name: install and start services

hosts: all

vars:

services:

- httpd

- smb

tasks:

- name: start services

service:

name: "{{ item }}"

state: started

loop: "{{ services }}" <---uses the defined var in the header for the loop

Using Loops on Multivalued Variables

* Multivalued variables can be used if they are presented as a list

Example:

(file named /vars/user-list)

users:

- username: linda

homedir: /home/linda

shell: /bin/bash

groups: wheel

- username: jim

homedir: /home/dir

shell: /bin/bash

groups: wheel

(playbook)

---

- name: create users in a loop

hosts: ansible1

vars\_files: /vars/user-list <----references the var file above

tasks:

- name: create users

user:

name: "{{ item.username }}"

state: present

groups: "{{ item.group }}"

shell: "{{ item.shell }}"

loop: "{{ users }}"

Using when to Run Tasks Conditionally

* Using when runs tasks conditionally based on a criteria (variables, files existing, etc.)
* Options for when:
* variable exists = variable is defined
* variable does not exist = variable is not defined
* first variable is present in list mentioned as 2nd = ansible\_distribution in distributions
* var is true, 1, or yes = variable
* var is false, 0, or no = not variable
* equal (string) = key == "value"
* equal (integer) = key == value
* less than = key < value
* less than or equal to = key <= value
* greater than = key > value
* greater than or equal to = key >= value
* not equal = key != value

Example:

- name: install apache on RedHat

yum:

name: httpd

state: latest

when: ansible\_facts['os\_family'] == "RedHat"

* Multiple testing conditions can be used using 'and' or 'or' and then can be grouped in parenthesis as well
* You can also loop over when tests and combine them with register

Using Handlers

* A handler is a task that is triggered and executed by a successful task
* To use handlers, you define a notify statement at the same level as task
* notify statement lists name of handler and handler is listed at the end of the play
* A handler only runs when the task that triggers it returns a changed status
* If a handler is triggered and a task later in the same play fails, the handler will not be executed on the host, this can be avoided using the force\_handlers: true in the play header or the generic ignore\_errors: true (playbook runs even after a task failure)

Example:

(after creating an index.html file)

- name: copy index.html

copy:

src: /tmp/index.html

dest: /var/www/html/index.html

notify:

- restart\_web

(end of play)

handlers:

- name: restart\_web

service:

name: httpd

state: restarted

Task Execution

* Tasks execute in order, when one fail the play stops & generates an error
* To stop a playbook for all hosts when a failed task is encountered, you can add any\_errors\_fatal in header or block
* Tasks generate 3 results: OK = success with no changes made to host to match desired state (2) CHANGED = success with changes made (3) FAILED

Specify Task Failure Conditions

* if a command runs on a managed node, Ansible reports 'success'. To change this use failed\_when condition

Example:

- name: run echo

command: echo hello

register: command\_result <----store result as a variable

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

* The fail module can also be used if when is being used to define the condition when a failure should occur (good for specifying a clear failure message)

Example:

fail:

msg: the command has failed

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

Managing Changed Status

* The changed status doesn't always mean something was changed, sometimes it just means a command was successfully run
* The changed\_when: false parameter in the task makes a task only able to return ok or failed

Using Blocks

* A block is a group of tasks which a when statement can be applied, essentially it means that if a single condition is true, multiple tasks execute
* To use this, insert a block: statement after a task name

Example: (the when statement will need to be in the same indent level as block statement

tasks:

- name: set up httpd

block:

- name: install httpd

yum:

name: httpd

state: present

Using Blocks with Rescue and Always Statements

* rescue statements execute if block tasks fail \*Sander says this is likely to be on exam
* always statements define tasks that should run no matter what happens \*Sander says this is likely to be on exam

File Module Manipulation Overview (modules)

copy = copy files to remote locations

fetch = fetch file from remote locations

file = manage files and preparations

acl = works with file system and ACLs

find = find files based on any property

lineinfile = manage lines in text files

blockinfile = manage blocks in text files

replace = replace strings in text files based on regex

synchronize = performs rsync-based synchronization tasks

stat = retrieve file or file system status

\*TIP\* ansible-doc command on modules to see "see also" section in case a module doesn't fit your needs

Managing File Attributes

* The stat module enables you to retrieve file status info and can help the file module
* stat returns a lot of attributes like permissions, if a path leads to a dir or file, etc.

Example:

---

- name: stat module tests

hosts: ansible1

tasks:

- stat:

path: /etc/hosts

register: st

- name: show value

debug:

msg: current vlaue of st var is {{ st }}

- fail:

msg: "unexpected file mode, should be 0640"

when: st.stat.mode != '0640'

Creating and Removing Files

* The file module can create new files/dirs, create links, remove files, and set permissions/ownership

Example:

- name: create dir

file:

path: /newdir

owner: ansible

group: ansible

mode: 770

state: directory

- name: create a file

file:

path: /newdir/newfile

state: touch

- name: remove everything we made

file:

path: /newdir

state: absent

Moving Files

* The three most relevant modules are (1) copy = copy to remote host, (2) fetch = retrieve from a remote host, (3) synchronization = rsync-like tasks

Example

- name: copy a file

copy:

src: /etc/hosts

dest: /tmp/

Managing SELinux (modules)

file = manage context on files but not in the SELinux policy (not persistent)

sefcontext = manage file context in SELinux policy

command = is required to run restorecon command after using sefcontext

selinux = manage current SELinux state

seboolean = manage SELinux booleans

Manage SELinux File Context

* File context defines what processes can work with the files
* Must be set in SELinux policy for persistence, following this you have to run restorecon to apply the policy
* You must have policycoreutils-python-utils installed

Example:

- name: set selinux context

sefcontext:

target: /tmp/selinux <----file created in other script

setype: httpd\_sys\_content\_t

state: present

notify:

- run: restorecon

handlers:

- name: run restorecon

command: restorecon restorecon -v /tmp/selinux

Using Jinja2 Templates

* A template is a config file w/vars and is generated on managed hosts based on requirements
* More structured and powerful than editing specific lines in files

Working with Simple Templates

data = sample text

comment = { # sample text # }

variable = {{ ansible\_facts['default\_xyz'] }}

expression = {% for myhost in groups['web']%}

* To use a template you must create a template file written in jinja2 and make sure it is included in an ansible playbook that uses the template module

Example:

- name: install vhost config file

template:

src: listing813.j2

dest: /etc/httpd/conf.d/vhost.conf

Applying Control Structure in Jinja2 Using for

* A control structure can be used to dynamically generate contents
* A for statement can be used to iterate over all elements that exist as the value of a variable

Example:

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

host\_port = {{ node }}: 8080

(% endfor %}

Using Conditional Statements with if

* An If statement iterates only if a variable contains a specific value or evaluates to a boolean true

Example:

{% if apache\_package == 'apache2' %}

Welcome to Apache2

{% else %}

Welcome to httpd

{% endif %}

Using Filters

* Filters are a way to perform an operation on the value of a template expression, such as a variable

{{ myvar | to\_json }} write contents of myvar in JSON formate

{{ myvar | to\_yaml }} write to yaml

{{ myvar | ip\_addr }} test myvar to see if it contains an IP address

Using Ansible Roles

* Ansible roles work with include files
* Different components of the role are stored in different subdirectories
* Role Directory structure:

defaults = default variable that may be overwritten by other variables

files = static files that are needed by role tasks

handlers = handlers for use in this role

meta = metadata such as dependencies, plus license and maintainer info

tasks = role task definitions

templates = Jinja2 templates

tests = optional inventory and a test.yml file to test the role

vars = variables that are not meant to be overwritten

Understanding Role Location

* Before making a role, you define where to store it

./roles = store roles in current project directory (takes precedence)

~/.ansible/roles = exist in current user home directory and available to current user only

/etc/ansible/roles = roles are stored to make them accessible to any user

/usr/share/ansible/roles = roles stored after they are installed from REM files

* To make a custom role, you just make the default role directory structure with a main.yml file at the expected location
* Use the ansible-galaxy init command to create all the directories including samples for you

Using Roles from Playbooks

* Roles are called like tasks in a playbook

Example:

---

- name: include roles

roles:

- role1

- role2

* Roles execute before any tasks unless included in a pre\_tasks section

Managing Role Dependencies

* Roles may have dependencies based on other roles

Example:

- role: apache

port: 8080

- role: mariadb <----using mariadb role in the production environment

when: environment == 'production'

Understanding File Organization Best Practices

* Use project directories in bigger environments
* Each project directory can have its own .cfg file, inventory, and playbooks
* At some point in size, var files and others may be added into sub directories
* At top level directory, make the main playbook, suggested name is site.yml
* Consider using different inventory files to differentiate between prod and test
* Use roles to standardize common tasks
* never include sensitive info in roles (use ansible vault)
* Use ansible-galaxy init to create role base structure and remove files/directories you don't use
* include additional info in role's README.md and maet/main.yml files
* Keep roles focused on specific functions
* Try making roles in a generic way so they can be multi-purpose

Using Ansible Galaxy Roles and Commands

* Many roles do not need to be created as they come with Ansible Galaxy
* Roles can be downloaded directly from galaxy website or via the ansible-galaxy command
* The ansible-galaxy search command can find roles based on criteria

--platforms = OS platform

--author = github username

--galaxy-tags = additional tags to filter by

* After finding a role, use ansible-galaxy info

Managing Ansible Galaxy Roles

* ansible-galaxy install command installs roles into ~/.ansible/roles, and the -p option can be used to specify a different location
* A requirements file can be used, it is a yaml file, use -r option, for example ansible-galaxy install -r roles/requirements.yml
* ansible-galaxy list is used to see current installed roles
* ansible-galaxy remove is used to delete roles

Using RHEL System Roles

* rhel-system-roles.<insert one of below>

kdump = configure the kdump crash recovery service

network = configure network interfaces

postfix = configure hosts as mail transfer agent

selinux = manage selinux settings

storage = configure storage

timesync = configure timesync

* Install a system role with sudo yum install rhel-system-roles
* The SELinux system role comes with a sample playbook file that can be copied and edited for use in /usr/share/doc/rhel-system-roles/selinux

Advanced Inventory Usage

* When environments get larger, host name patterns, dynamic inventory, and multiple inventory files are helpful
* You cannot just specify an IP in a playbook and not in an inventory (will not work)
* Using wildcards can be used instead of groups or other predefined sets of hosts, for example ansible -m ping 'ansible\*' will ping all hosts starting with ansible
* When using wildcards. the pattern patter is evaluated against IPs and groups as well so It may match both a host and group
* Multiple hosts can also be specified using comma separated values
* logical conditions can be used, ex. web & file = hosts in web AND file groups, web,!webserver1 = hosts in web but NOT webserver1

Configuring Dynamic Inventory

* Dynamic inventory = a script that can be used to detect whether new hosts have been added to the managed environment
* can be community provided or custom, but must have execute permissions
* to view hosts in dynamic inventory, run ansible -i <inventory> all --list-hosts

Using the ansible-inventory Command

* To view inventories in a graph format, use the command ansible-inventory -i <filename> --graph (file must be executable)
* When working with multiple inventories, multiple -i parameters can be used, ex. ansible-inventory -i inventory -i listing100.py --list \*the -i parameter can also be used to specify a directory that contains all inventories

Optimizing Parallel Task

* Parallel task execution = manage the number of hosts on which tasks are executed
* Serial task execution = make sure that all tasks are executed on a host/group before proceeding to next set

Managing Parallel Task Execution

* Processing is done on managed nodes, if they do not have a python stack however, Ansible limits connections to 5 concurrently by default
* use the forks parameter in cfg file or -f option in ansible-playbook to change, changing to 100 or so is fine
* By default, Ansible runs a task on all hosts before going to the next task. This can be changed using the serial keyword in the header, ex. serial: 3 <----run all tasks on 3 hosts then go to the next 3

Including and Importing Files

* When playbooks get very large, splitting them up based on management tasks is recommended
* When content is included, it is dynamically processed at the moment ansible reaches that content
* Files can be included and imported at different levels: (1) roles (2) playbooks (3) tasks (4) variables

Importing Playbooks

* Common setup: one master playbook that uses additional playbooks. Use the import\_playbook module

Example:

---

- name: run a task

hosts: all

tasks:

- debug:

msg: running task 1

- name: importing a playbook

import\_playbook: listing100.yml <----this is importing a playbook at the play level

Importing and Including Task Files

* You can also import tasks instead of complete playbooks
* the import\_tasks module will import tasks statically for execution
* when you use include\_tasks they are dynamically included when needed )recommended when task file is used in a conditional statement)
* When using import\_tasks:
* loops cannot be used
* if a variable is used to specify the name of the file to import, it cannot be a host or group variable
* when using a when statement on the entire import\_tasks file, the conditional statements are applied to each task involved
* When using include\_tasks:
* when using ansible-playbook --list-tasks command, tasks that are in the included task are not displayed
* you cannot use ansible-playbook --start-at-task to start a playbook from an included file
* you cannot use a notify statement in the main playbook to trigger a handler that is in the included task file

Managing Ansible Errors and Logs

* Check mode = shows what changes would take place if the play was run, uses the -C option, ansible-playbook -C (this does not work on playbooks with conditionals since they depend on changes actually being made)
* check mode: yes in the header forces the playbook to run in check mode only

Understanding Output

* In the output of the ansible-playbook command, it includes:
* indicator of the play that is started
* if not disabled, the Gathering Facts task that is executed for each play
* each individual task, including its name
* play recap summarizing the results:
  + ok = current state matches desired state
  + changed = a change was made to implement a desired state
  + unreachable = target could not be reached
  + failed = task contained error and couldn't execute
  + skipped = due to not matching a conditional, the task skipped
  + rescued = main task in a block failed and rescue action was used
  + ignored = target host was ignored due to use of ignore\_errors in playbook

Optimized Command Output Error Formatting

* add stdout\_callback = debug and stdout\_callback = error to .cfg file to make errors easier to read

Logging to Files

* You can set the log\_path = parameter to have ansible write errors to a file

Running Task by Task

* To run a playbook task-by-task asking for confirmation along the way, use ansible-playbook --step
* To start at a specific task, use ansible-playbook --start-at-task="<name>"
* Use ansible-playbook --list-tasks for a list of tasks in the playbook (but they must be named to be seen)

Using Modules to Troubleshoot

* debug = writes debug info (useful for checking variables)
* uri = tests answer coming from any URL
* fail = uses a when conditional to tell when a module should be considered failing
* script = allows execution of a shell script on managed host
* stat = gathers status info about files
* assert = tests whether the expected result is present and otherwise fails

Debug Module

* Works with 2 arguments (1) msg = print a message (2) var = used to print the value of a variable
* Variables in msg must use {{ varname }} but in var they can be referred to without the {{ }}, just the var name

URI Module

Example:

---

- name: test webserver access

hosts: all

tasks:

- name: connect to server

uri:

url: <http://ansible1.com>

return\_content: yes <----capture server content

register: servercontent <----store the content as variable

failed\_when: " 'Welcome' not in servercontent.content" <----define failure

- debug:

var: servercontent.content <----show contents of the stored variable

Using Stat Module

* Useful in combination with register

Example:

tasks:

- stat:

path: /tmp/statfile

register: stat\_out

- fail:

msg: "/tmp/statfile owner not as expected"

when: stat\_out.state.pw\_name != 'root'

Assert Module

* Works with a that option to perform a conditional action
* if any conditions in that option fail = task fails, if they are all true then they succeed
* Based on the success or failure of a task, the success\_msg or fail\_msg prints

Using Tools

* tags are used as a label applied to a task or other item like a block or play
* using ansible-playbook --tags or --skip-tags can specify which tasks to execute
* tags can be used multiple times to group tasks

Example:

tasks:

- name: install the services

- service:

name: "{{ item }}"

state: started

enabled: yes

loop: "{{ services }}"

tags:

- install

ansible-playbook --tags "install" example.yml <----you can also use --list-tags instead to view them

Troubleshooting Common Scenarios

* Always make sure:
* you have an IP network connection
* SSH is accessible on remote host
* Python is installed on control host
* privilege escalation is set up
* check your host and inventory file
* use the ping module for quick checks

Analyzing Authentication Issues

* Settings that play a role in authentication on remote host:
* The remote\_user setting in the cfg file determines which user account to use on the managed node
* SSH keys need to be configured for the remote\_user to enable smooth authentication
* The become parameter needs to be set to true
* the become\_user needs to be set to root
* sudo needs to be set correctly

Managing Software with Ansible

* Software management modules:
* yum = manage software on RHEL/CentOS
* dnf = manage software on Fedora/RHEL
* apt = manage software on Ubuntu
* packge = manage software on any Linux distro
* yum\_repository = manage repos
* package\_facts = include package facts in the ansible facts

Configuring Repository Access

* To set up access to a repo, the yum\_repository module is provided
* Arguments for yum\_repository:

name: name of repo

description: provide details

file: identify the .repo file created in /etc/yum.repos.d stored on managed node

baseurl: URL to access repo

gpgcheck: whether to check the files argument against GPG key

Example:

---

- name: set up repo access

hosts: all

tasks:

- name: connect to repo

yum\_repository:

name: example repo

description: RHCE example

file: examplerepo

baseurl: <ftp://control.example.com/repo>

gpgcheck: no

Managing Software with Yum

* Yum arguments:
* name: name of package
* state: present or installed will (install), latest (update and/or install if necessary), absent/removed (delete)
* package groups can be supplied as a name using name:'@packageorgroup'
* When ansible gathers facts, packages are not included unless the package\_facts module is used

Modules for Users/Groups

* user: manage users and their properties
* group: manage groups and their properties
* pamd: manage advanced auth config through PAM
* known\_hosts: manage SSH known hosts
* authorized\_key: copies user authorized key to a managed host
* lineinfile: modify config files

Users and Groups

* The group argument in user module can specify the primary group of that user
* The groups (plural) argument is used to specify additional groups for a user
* When using groups argument for existing users, make sure to use append argument or it will overwrite other groups

Example:

---

- name: create user and a group

hosts: ansible1

task:

- name: setup group account

group:

name: students

state: present

- name: setup user account

user:

name: anna

create\_home: yes

groups: wheel, students

append: yes

generate\_ssh\_keys: yes

ssh\_key\_bits: 2048

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

Managing Sudo

* Managing sudo can be done suing a template module to create a sudo config file in /etc/sudoers.d
* Or you can use the lineinfile module to edit the /etc/sudoers

Managing SSH Connections

* The authorized\_key module along with generate\_ssh\_key argument to automatically generate SSH keys

Setting Up SSH User Keys

* You can use the copy module to copy the public key from control to managed host
* If you want to manage multiple keys in the authorized\_keys file, the authorized\_key module may be better

Example:

- name: copy auth keys for ansible user

authorized\_key:

user: ansible

state: present

key: "{{ lookup('file', 'home/ansible/.ssh/id\_rsa.pub') }}"

Managing Encrypted Passwords

* The password argument in the user module cannot accept unencrypted strings
* An external utility should be used to generate the string and store it in a variable securely, such as ansible vault
* You can create the variable before making the user or after using the command to create the variable in the playbook and then the register argument to store the result
* To generate the variable beforehand, you can use ansible localhost -m debug -a "msg = {{ 'password' | password\_hash('sha512', 'myrandomsalt') }}" then copy the output and paste in playbook

Example:

vars:

password: sdiasnusahf98123ihfncn <----hash copied from step above

- name: create user

user:

name: anna

password: "{{ password }}"

Using an Alternative Approach

* Another 'dirty' method is is the command echo password | passwd --stdin

Example:

vars:

password: password123

user: anna

tasks:

- name: config users {{ user }}

user:

name: "{{ user }}"

groups: wheel

append: yes

state: present

- name: set pw for {{ user }}

shell: 'echo {{ password }} | passwd --std-in {{ user }}'

Managing Processes and Tasks

* service: works for multiple init systems; generic module for services
* service\_facts: gather facts for services started by BSD init, upstart, or systemd
* cron: Linux cron scheduler
* at: Linux at scheduler for one-time jobs
* systemd: manage systemd specific service properties (must overlap with service module)
* reboot: self-explained

Managing Systemd Services

* If systemd specifics need to be addressed the systemd module must be used, for example daemon\_reoload and mask

Example:

systemd:

name: httpd

enabled: yes

masked: no

daemon\_reload: yes

Managing Cron Jobs

* The cron module can (1) write jobs in a user's crontab (2) write a job to /etc/crontab or under /etc/crn.d directory, (3) pass job directly to anacron to be run once an hour/day/week/month/year

Example:

cron:

name: "un fstrim"

minute: "5"

hour: "4, 19"

job: "fstrim"

user: ansible (or who you want the job run as)

state: present | absent (choose one) to add or remove the job

Managing at Jobs

* command: the command that needs to be executed
* units: min/hour/day/week of task execution
* count: the # of units to execute the task at
* script\_file: name of script to execute
* state: uses added or deleted to add/delete a specific command
* unique: set to yes to ensure a specific job is started once only
* count units: how far from now to execute tasks (eg count:5 , units: minute = 5 minutes from now)

Rebooting Managed Hosts

* test\_argument: needs to be applied to specify an arbitrary command for ansible to run successfully on host to indicate it is back up after reboot
* Arguments related to timeouts:
* connect\_timeout: maximum seconds to what wait for successful connection before retrying
* post\_reboot\_delay: number of seconds to wait after reboot command before trying to validate
* pre\_reboot\_delay: number of seconds to wait before actually issuing reboot
* reboot\_timeout: max number of seconds to wait for rebooted machine to respond to test command

Example:

reboot:

msg: reboot initiated by ansible

test\_command: whoami

- name: print message to show host is back

debug:

msg: successfully rebooted

Discovering Storage-Related Facts

* parted: manage partitions in a scripted way
* lvg: manage LVM volume groups
* lvol: manage LVM logical volumes
* filestystem: manage filesystems on storage devices
* mount: mounts storage devices that have file systems configured
* vdo: interfaces the new VDO storage layer

Using Storage-Related Facts

* ansible\_devices: shows all available devices
* ansible\_device\_links: show storage devices from different perspectives (eg. how to access devices)
* ansible\_mounts: show mount points and device info for what is mounted

These facts can be viewed with ansible -m setup command, eg. ansible ansible2 -m setup -a 'filter=ansible\_device'

Using Storage-Related Facts

* Using when statements can be good for seeing if a device exists but using assert module can also print an error message if condition is met

Example:

assert:

that:

- "ansible\_facts['device']['{{ disk\_name }}'] is defined"

fail\_msg: second hard disk not found

Creating Partitions Using Parted

* name: required for GPT partitions
* label: specify partition type (either mdos or gpt)
* device: device to make a partition on
* number: partition number
* state: present=create, absent=remove
* part\_start: starting position for the partition
* part\_end: ending position for partition
* flags: set specific partition properties such as LVM type

Example:

parted:

name: files

label: gpt

device: /dev/sdb

number: 1

state: present

part\_start: 1MB

part\_end: 2GiB

Managing Volume Groups and LVM Logical Volumes

* lvg module manages volume groups and the lvol module manages logical volumes
* For lvol:
* lv: specifies the name of the logical volume
* pvs: indicates the comma separated list of pvs (make sure pv has lvm option set)
* resizefs: the underlying filesystem must be resized automatically when lv is resized
* size: size of lv
* snapshot: specifies the name if lv is a snapshot
* vg: indicates the name of vg in which the lv should be created

Example:

lvg:

vg: vgdata

pesize: "8"

pvs: /dev/sdb1

lvol:

lv: lvdata

size: 100%FREE

vg: vgdata

Creating and Mounting File Systems

* After making a storage setup, put a filesystem on it using the filesystem module:
* dev: indicates the block device name
* fstype: specifies the filesystem type (ext4, xfs, etc.)
* opts: identifies options based on the underlying mkfs command
* resizefs: extends the filesystem if set to yes
* After making filesystem, mount it using the mount module:
* fstype: type of filesystem
* path: directory to mount fs on
* src: device that should be mounted
* state: current state, mounted=mount device now, present=puts it into fstab but doesn't mount immediately

Example:

mount:

src: /dev/vgdata/lvdata

fstype: xfs

state: mounted

path: /mydir

Configuring Swap Space

* To set up swap space you first format a device as swap space and mount it
* This is done using the filesystem module and then command module to run the swapon command

Example:

- name: setup swap space

block:

- name: make swap fs

filesystem:

fstype: swap

dev: /dev/sdb1

- name: activate swap

command: swapon /dev/sdb1

General Exam Tips

* Read ALL questions before starting
* Make a task list for each question before working on it
* When done with all questions, take a break and come back before checking them
* There is no way a solution MUST be, just get it up and running