REDHAT LINUX

RHCE

Version : RHEL8.X

Outline of the course:

1. Introduce Ansible
2. Deploy Ansible
3. Implement playbooks
4. Manage variables and facts
5. Implement task control
6. Deploy files to managed hosts
7. Manage large projects
8. Simplify playbooks with roles
9. Troubleshoot Ansible

Ansible

Agenda

- Ansible Introduction
- Use Cases
- Common Terminology
- Ansible Architecture
- Ansible Installation

Ansible Introduction

Ansible is an open source **IT automation tool**.

It can configure systems, deploy software, and orchestrate more advanced IT tasks such as continuous deployments or zero downtime rolling updates.

Ansibles enables you to manage and control multiple servers from one central location

It runs on UNIX like operating systems and configure both UNIX like operating systems and MS Windows.

Ansible Use Cases

- Configuration Management
- Provisioning
- Application Deployment/Orchestration
- Compliance Checks (firewall rules, enable/disable services)

Ansible Features

Agentless – Which means there is no kind of software or any agent managing the node like other solution such as puppet and chef.

Python Based – Built on top of python, which is fast and one of the robust programming languages in today's world.

SSH – Very simple password less network authentication protocol which is secure. So, your responsibility is to copy this key to the client

YAML Syntax- Human-readable and quite easy to follow

Ansible

Push architecture – Push the necessary configurations to clients. All you have to do is, write down those configurations (playbook) and push them all at once to the nodes.

Ansible Terminology

- Control Node

Any system upon which Ansible is installed and which has access to the required configuration files and playbooks to manage remote systems (managed hosts) is called a control node.

- Managed Nodes

- Host Inventory

Managed hosts are defined in the inventory. Host patterns are used to reference managed hosts defined in an inventory.

Ansible Terminology

- Task

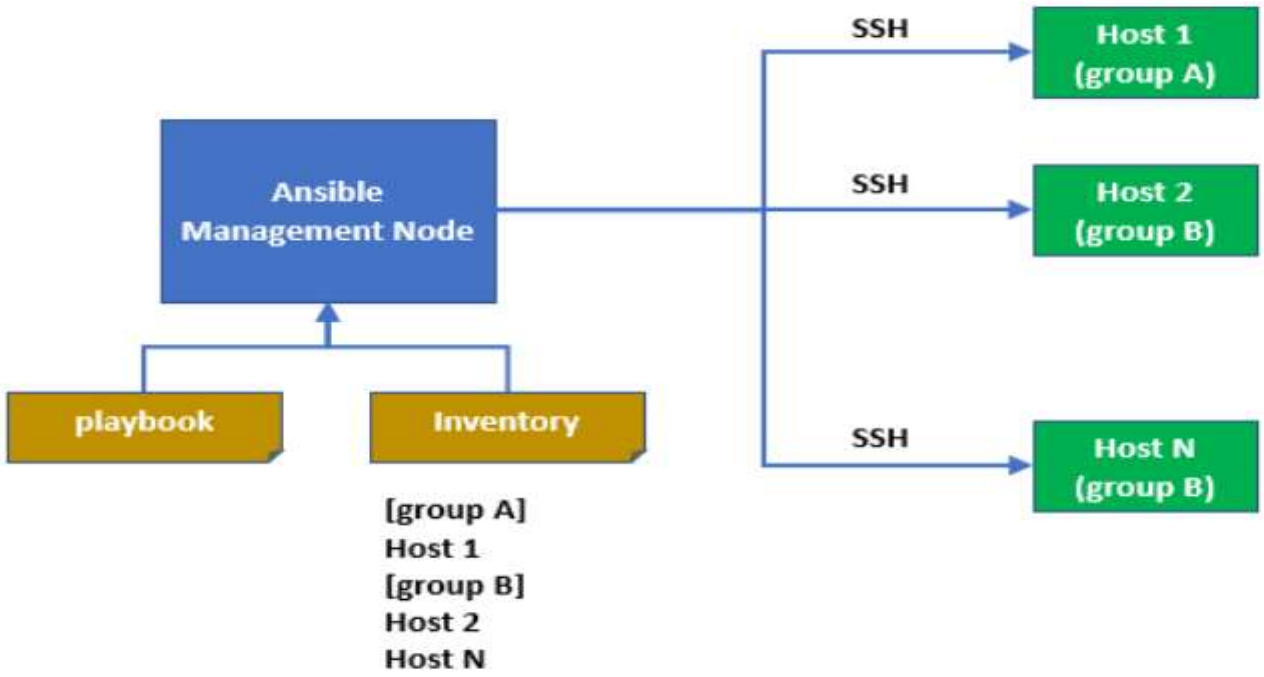
- Play

A play is an ordered set of tasks run against hosts selected from your inventory.

- Playbook

A playbook is a text file written in YAML format containing a list of one or more plays to run in a specific order it is normally saved with the extension yml.

Ansible workflow



Ansible Installation

Enable EPEL repository

Extra Packages for Enterprise Linux (or EPEL) is a Fedora Special Interest Group that creates, maintains, and manages a high quality set of additional packages for Enterprise Linux, including, but not limited to, Red Hat Enterprise Linux (RHEL), CentOS, Oracle Linux (OL) etc.

https://dl.fedoraproject.org/pub/epel/
https://dl.fedoraproject.org/pub/epel/epel-release-latest-8.noarch.rpm

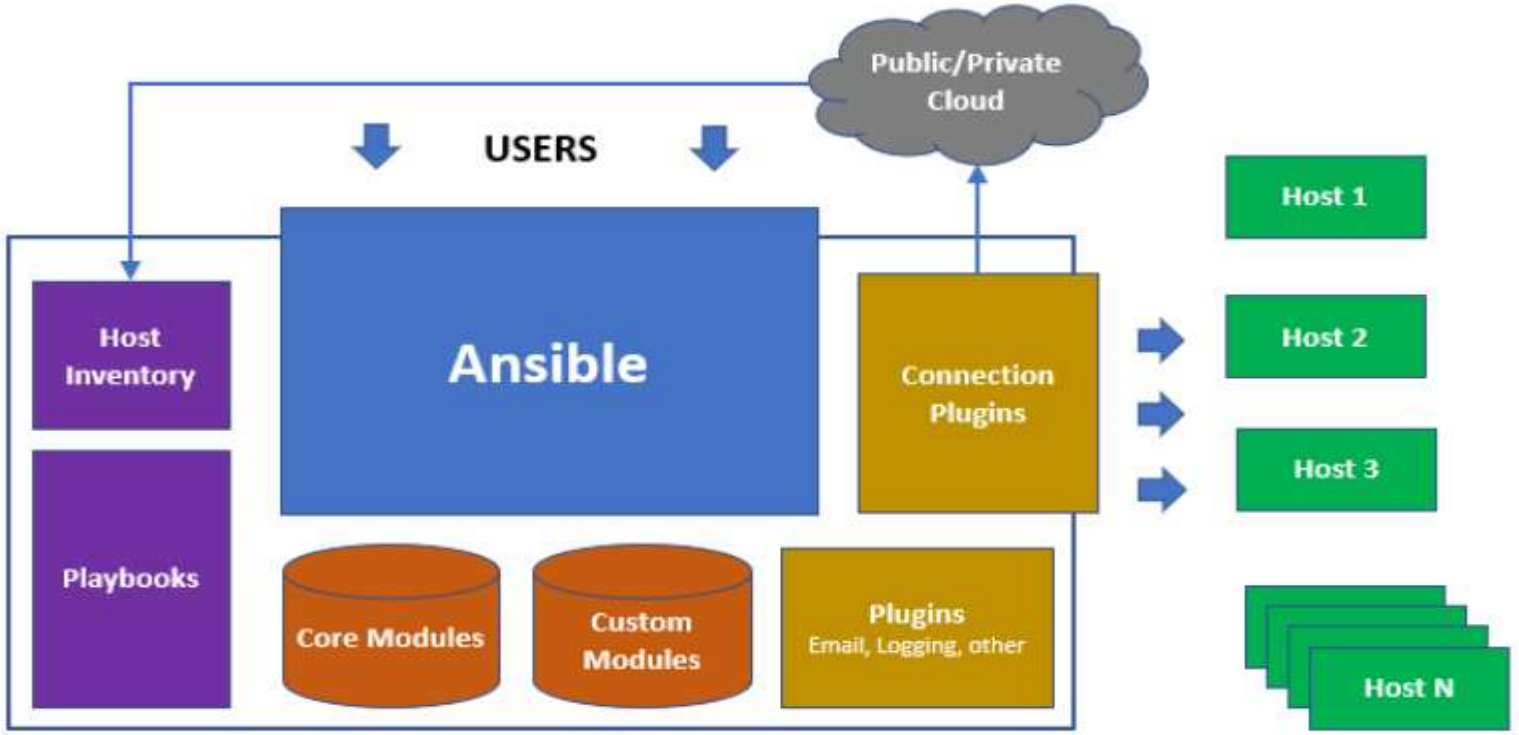
Install ansible and check the version.

Ansible Deployment

Agenda

- Configure Ansible host inventory
- Introduction to Ansible modules
- Running ad hoc Ansible commands

Ansible Deployment



Ansible Deployment

Host Inventory (/etc/ansible/hosts)

An inventory defines a collection of hosts that Ansible will manage. These hosts can also be assigned to groups, which can be managed collectively.

Inventory file is a text file that specifies the managed hosts that Ansible targets. It can be written in different formats, including INI-style or YAML. The INI-style format is very common.

Copy the /etc/ansible/hosts file in the directory from where ansible commands are run.

Ansible Deployment

Host Inventory

- An INI-style static inventory file is a list of host names or IP addresses of managed hosts.
- Normally we organize managed hosts into host groups.
- Host groups allow you to more effectively run Ansible against a collection of systems.
- In this case, each section starts with a **host group** name **enclosed in square brackets ([])**. This is followed by the host name or an IP address for each managed host in the group, each on a single line.
- Hosts can be in multiple groups.

Ansible Deployment

Host Inventory (/etc/ansible/hosts)

```
[webservers]
alpha.example.org
beta.example.org
192.168.1.100
192.168.1.110


[dbservers]
db01.intranet.mydomain.net
db02.intranet.mydomain.net
10.25.1.56
10.25.1.57
```

Ansible Deployment

Defining Nested Groups

Ansible host inventories can include groups of host groups. This is accomplished by creating a host group name with the **:children suffix**.

The following example creates a new group called north-america, which includes all hosts from the usa and canada groups.

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

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

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

Ansible Deployment

Host Specifications with Ranges:

User can specify ranges in the host names or IP addresses to simplify Ansible host inventories.

```
[mailservers]

mail[01:04].intranet.mydomain.net
10.25.1.[0:05]
```

```
[root@host2 Desktop]# ansible -i ./hosts_custom --list-hosts mailservers
  hosts (10):
    mail01.intranet.mydomain.net
    mail02.intranet.mydomain.net
    mail03.intranet.mydomain.net
    mail04.intranet.mydomain.net
    10.25.1.0
    10.25.1.1
    10.25.1.2
    10.25.1.3
    10.25.1.4
    10.25.1.5
```

Ansible Deployment

Verifying the Inventory:

- use below ansible command to verify client machine's presence in the inventory.
- **all/ungrouped**

```
[root@host2 ansible]#
[root@host2 ansible]# ansible ungrouped --list-hosts
  hosts (1):
    green.example.com
[root@host2 ansible]#
[root@host2 ansible]# ansible webservers --list-hosts
  hosts (4):
    alpha.example.org
    beta.example.org
    192.168.1.100
    192.168.1.110
[root@host2 ansible]#
```

Ansible Deployment

Verifying the Inventory:

How to use custom inventory file

```
[root@host2 ~]#
[root@host2 ~]# pwd
/root
[root@host2 ~]# ls -la hosts_custom
-rw-r--r--. 1 root root 1016 Nov  7 14:34 hosts_custom
[root@host2 ~]#
[root@host2 ~]# ansible -i hosts_custom dbservers --list-hosts
  hosts (5):
    db01.intranet.mydomain.net
    db02.intranet.mydomain.net
    10.25.1.56
    10.25.1.57
    10.25.1.58
[root@host2 ~]#
```

Ansible Deployment

Lab exercise

❑ Ansible Inventory

- Guided Exercise

S.No	HOSTNAME	PURPOSE	LOCATION	ENVIRONMENT
1	client1.example.com	Web Server	Delhi	Development
2	client2.example.com	Web Server	Delhi	Testing
3	client3.example.com	Web Server	Lucknow	Production
4	client4.example.com	Web Server	Noida	Production

Ansible Deployment

Configuring Ansible

Copy the ansible.cfg file in the directory from where ansible commands are run.

```
[testuser@host2 ~]$ ansible --version
ansible 2.9.15
  config file = /home/testuser/ansible.cfg
  configured module search path = ['/home/testuser/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python3.6/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 3.6.8 (default, Jun 26 2020, 12:10:09) [GCC 8.3.1 20191121 (Red Hat 8.3.1-5)]
[testuser@host2 ~]$
```

Ansible Deployment

Escalating Privileges

For security and auditing reasons, Ansible connects to remote hosts as an unprivileged user before escalating privileges to get administrative access as root. This can be set up in the ***[privilege_escalation]*** section of the ***ansible.cfg*** file.

```
[privilege_escalation]
become=True
become_method=sudo
become_user=root
become_ask_pass=False
```

On the managed host add the user into the *sudoers* file.

Ansible Deployment

Ansible Command-line Options

CONFIGURATION FILE DIRECTIVES	COMMAND-LINE OPTION
inventory	-i
remote_user	-u
become	--become, -b
become_method	--become-method
become_user	--become-user
become_ask_pass	--ask-become-pass, -K

Ansible Deployment

Ansible modules

Modules (also referred to as “task plugins” or “library plugins”) are discrete units of code that can be used from the command line or in a playbook task.

Ansible executes each module, usually on the remote managed node, and collects return values.

Ansible Deployment

Ansible modules

- ping
- command
- shell
- copy
- service
- Many More ……..

[root@host2 ~]# ansible-doc -l
[root@host2 ~]# ansible-doc command
> COMMAND (/usr/lib/python3.6/site-packages/ansible/modules/commands/command.py)

Ansible Deployment

Running AD HOC Commands With Ansible

- An ad hoc command is a way of executing a single Ansible task quickly. They are simple, online operations that can be run without writing a playbook.
- Ad hoc commands are useful for quick tests and changes. For example, one can use an ad hoc command to make sure that a certain line exists in the /etc/hosts file on a group of servers. One could use another ad hoc command to efficiently restart a service on many different machines, or to ensure that a particular software package is up-to-date.
- Use the ansible command to run ad hoc commands:

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

Ansible Deployment

Running Ad-hoc commands

```
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible -i ./hosts_test myservers -m ping
192.168.1.3 | SUCCESS => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": false,
    "ping": "pong"
}
[testuser@host2 ~]$
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible -i ./hosts_test myservers -m command -a "ls -lrt"
192.168.1.3 | CHANGED | rc=0 >>
total 0
drwxrwxr-x. 2 testuser testuser 6 Nov  7 17:46 client
-rw-rw-r--. 1 testuser testuser 0 Nov  7 17:52 client_file
[testuser@host2 ~]$
[testuser@host2 ~]$
```

Ansible Deployment

Running Ad-hoc commands

```
[testuser@host2 ~]$ ansible -i ./hosts_test myservers -m copy -a 'src=./hosts_test  dest=/home/testuser/host_test_managed'
192.168.1.3 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": true,
    "checksum": "469b4b856d79431304f2b0c77ba593e5099d0446",
    "dest": "/home/testuser/host_test_managed",
    "gid": 1017,
    "group": "testuser",
    "md5sum": "333ac6f4837ba10c282719f9e1ba54f1",
    "mode": "0664",
    "owner": "testuser",
    "secontext": "unconfined_u:object_r:user_home_t:s0",
    "size": 1047,
    "src": "/home/testuser/.ansible/tmp/ansible-tmp-1604752364.267245-3476-158421809096356/source",
    "state": "file",
    "uid": 1017
}
```

Ansible Deployment

Running Ad-hoc commands

```
[testuser@host2 ~]$ ansible -i ./hosts_test myservers -m user -a 'name=tempuser uid=2001 state=present'
192.168.0.6 | CHANGED => {
    "ansible_facts": {
        "discovered_interpreter_python": "/usr/libexec/platform-python"
    },
    "changed": true,
    "comment": "",
    "create_home": true,
    "group": 2001,
    "home": "/home/tempuser",
    "name": "tempuser",
    "shell": "/bin/bash",
    "state": "present",
    "system": false,
    "uid": 2001
}
```

Ansible Deployment

Shell vs. Command module

In the most use cases both modules **command** and **shell** lead to the same goal. Here are the main differences between these modules.

•With the Command module the command will be executed without being proceeded through a shell. As a consequence some variables like $HOME are not available. And also stream operations like <, >, | and & will not work.

•The Shell module runs a command through a shell, by default /bin/sh. This can be changed with the option executable. Piping and redirection are here therefor available.

•The command module is more secure, because it will not be affected by the user's environment.

Ansible Deployment

command vs shell module

```
[testuser@host2 ~]$ ansible -i hosts_test myservers  -m command -a 'ls -l | grep client'
192.168.1.3 | FAILED | rc=2 >>
client:
total 0ls: cannot access '|': No such file or directory
ls: cannot access 'grep': No such file or directorynon-zero return code
[testuser@host2 ~]$
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible -i hosts_test myservers  -m shell  -a 'ls -l | grep client'
192.168.1.3 | CHANGED | rc=0 >>
drwxrwxr-x. 2 testuser testuser    6 Nov  7 17:46 client
-rw-rw-r--. 1 testuser testuser    0 Nov  7 17:52 client_file
[testuser@host2 ~]$
```

Ansible Deployment

Ansible idempotency

Idempotence is a term given to certain operations in mathematics and computer science whereby:

An action which, when performed multiple times, has no further effect on its subject after the first time it is performed e.g. Multiplication by one.

For Ansible it means after 1st run of a playbook to set things to a desired state, further runs of the same playbook should result in 0 changes. In simplest terms, idempotency means you can be sure of a consistent state in your environment.

Copy same file twice on remote.
Try modifying the files and try again.

Ansible Playbook

Agenda:

1. **Ansible plays**
2. **YAML files**
3. **Implement Ansible playbooks**
4. **Write and execute a playbook**

Ansible Playbooks

- Ad hoc commands can run a single, simple task against a set of targeted hosts as a one-time command.
- A play is an ordered set of tasks run against hosts selected from your inventory.
- A playbook is a text file that contains a list of one or more plays to run in order.
- Playbooks are one of the core features of Ansible and tell Ansible what to execute.
- Ansible Playbooks offer a repeatable, re-usable, simple configuration management and multi-machine deployment system, one that is well suited to deploying complex applications.

Ansible Playbooks

- If user needs to execute a task with Ansible more than once, write a playbook and put it under source control. Then user can use the playbook to push out new configuration or confirm the configuration of remote systems

Ansible Playbooks (YAML)

A playbook is a text file written in YAML format, and is normally saved with the extension yml. The playbook uses indentation with space characters to indicate the structure of its data.

Basic rules.

• Data elements at the same level in the hierarchy (such as items in the same list) must have the same indentation.

• Items that are children of another item must be indented more than their parents. You can also add blank lines for readability.

Important:
Only the space character can be used for indentation; tab characters are not

Ansible Playbooks (YAML)

Sample YAML file showing Single play

```yaml
---
-
 name: This is my test Playbook
 hosts: myservers
 tasks:
          - name: First task check dirs present in home
            shell: "ls -lrt"
            register: information

          - name: This is my Second task to print the output
            debug: msg="{{information.stdout}}"
...
```

Ansible Playbooks

Syntax Verification

```
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-playbook -i ./hosts_test --syntax-check user_creation.yml

playbook: user_creation.yml
[testuser@host2 ~]$
```

Ansible Playbooks

Dry Run of the playbook

```
[testuser@host2 ~]$ ansible-playbook -i ./hosts_test -C user_creation.yml

PLAY [myservers] ***************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************
ok: [192.168.0.6]

TASK [User Creation Task] *******************************************************************************************
changed: [192.168.0.6]

PLAY RECAP *********************************************************************************************************
192.168.0.6                : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0
```

Ansible Playbooks

Writing playbook having multiple plays

Manage variables and facts

Agenda:

- **Variables**
- **Managing Secrets**
- **Managing Facts**

Manage variables and facts

Introduction to Ansible Variables:

Ansible supports variables that can be used to store values that can then be reused throughout files in an Ansible project. This can simplify the creation and maintenance of a project and reduce the number of errors.

Variables provide a convenient way to manage dynamic values for a given environment in your Ansible project.

Variable Naming Convention:

Variable names must start with a letter, and they can only contain letters, numbers, and underscores.

Manage variables and facts

Introduction to Ansible Variables:

Usage:

- copy/remove files
- Packages to install
- Services to restart
- Creating new Users
- Archives to retrieve from the internet

Manage variables and facts

Introduction to Ansible Variables:

Scope:

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

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

Manage variables and facts

Defining Variables in Playbooks
One common method is to place a variable in a vars block at the beginning of a playbook.
Keyword **vars:**

Using Variables in Playbooks
Once variables have been declared, administrators can use those variables in tasks. Variables are referenced by placing the variable name in **double curly braces {{ }}**.
Ansible substitutes the variable with its value when the task is executed.

Passing Variables from command line:

ansible-playbook -i hosts_test -e "source=task_failure_2.yml" variable.yml

Manage variables and facts

Ansible facts:

Ansible facts are variables that are automatically discovered by Ansible on a managed host. Facts contain host-specific information that can be used just like regular variables in plays, conditionals, loops, or any other statement that depends on a value collected from a managed host.

Every play runs the ***setup*** module automatically before the first task in order to gather facts. This is reported as the Gathering Facts.

We can use ***debug*** module to print the value of the ***ansible_facts*** variable.

Manage variables and facts

Ansible facts:

Some of the facts gathered for a managed host might include:

- The kernel version
- The network interfaces
- The IP addresses
- The version of the operating system
- The number of CPUs
- The available or free memory
- The available disk space.

Sysadmins can take decisions using the facts during the automation.

Which system needs kernel upgrade.
Which system needs is running out of space.

Manage variables and facts

Ansible facts:

adhoc command method

[testuser@host2 ~]$ **ansible -i ./hosts_test myservers -m setup**

[testuser@host2 ~]$ **ansible -i ./hosts_test myservers -m setup -a "filter=ansible_memory_mb“**

[testuser@host2 ~]$ **ansible -i ./hosts_test myservers -m setup -a "filter=ansible_devices“**

Manging Ansible facts using playbook

Manage variables and facts

Ansible Vault (Managing secrets)

Ansible may need access to sensitive data such as passwords or API keys in order to configure managed hosts. Normally, this information might be stored as plain text in inventory variables or other Ansible files. In that case, however, any user with access to the Ansible files or a version control system which stores the Ansible files would have access to this sensitive data. This poses an obvious security risk.

Ansible Vault, which is included with Ansible, can be used to encrypt and decrypt any structured data file used by Ansible. To use Ansible Vault, a command-line tool named ***ansible-vault*** is used to create, edit, encrypt, decrypt, and view files. Ansible Vault can encrypt any structured data file used by Ansible.

Manage variables and facts

Ansible Vault (Managing secrets)

Creating encrypted playbooks:

```
[testuser@host2 vault]$ ansible-vault create  playbook_encrypted.yml
New Vault password:
Confirm New Vault password:
[testuser@host2 vault]$
[testuser@host2 vault]$ ansible-vault view  playbook_encrypted.yml
Vault password:
[testuser@host2 vault]$ ansible-vault edit  playbook_encrypted.yml
Vault password:
[testuser@host2 vault]$
```

Manage variables and facts

Ansible Vault (Managing secrets)

Creating encrypted playbooks (using password file)

[testuser@host2 vault]$ ***ansible-vault create --vault-password-file ./passwd playbook_encrypted.yml***
[testuser@host2 vault]$
[testuser@host2 vault]$ ***ansible-vault view --vault-password-file ./passwd playbook_encrypted.yml***

Manage variables and facts

Ansible Vault (Managing secrets)

Encrypting existing playbooks

[testuser@host2 vault]$ cat test.yaml

-
hosts: myservers

testuser@host2 vault]$ **ansible-vault encrypt test.yaml**
New Vault password:
Confirm New Vault password:
Encryption successful
[testuser@host2 vault]$
[testuser@host2 vault]$ cat test.yaml
$ANSIBLE_VAULT;1.1;AES256

Manage variables and facts

Ansible Vault (Managing secrets)

Encrypting existing playbooks

testuser@host2 vault]$ **ansible-vault encrypt test.yaml**
New Vault password:
Confirm New Vault password:
Encryption successful
[testuser@host2 vault]$
[testuser@host2 vault]$ **ansible-vault encrypt playbook_1.yaml --output=playbook_1_encrypt.yaml**
New Vault password:
Confirm New Vault password:
Encryption successful
[testuser@host2 vault]$

Manage variables and facts

Ansible Vault (Managing secrets)

Decrypting playbooks

[testuser@host2 vault]$ **ansible-vault decrypt playbook_1_encrypt.yaml --output=playbook_1_DEcrypt.yaml**
Vault password:
Decryption successful
[testuser@host2 vault]$

Changing password of playbook

[testuser@host2 vault]$ **ansible-vault rekey playbook_1_encrypt.yaml**
Vault password:
New Vault password:
Confirm New Vault password:
Rekey successful

Manage variables and facts

Ansible Vault (Managing secrets)

How to validate an encrypted playbook:

```
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-playbook -i hosts_test --syntax-check --vault-id @prompt plybook_2.yml
Vault password (default):

playbook: plybook_2.yml
[testuser@host2 ~]$
[testuser@host2 ~]$
```

Manage variables and facts

Ansible Vault (Managing secrets)

How to play an encrypted playbooks:

```
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-playbook -i ./hosts_test --vault-id @prompt ./vault/user_encrpt.yml
Vault password (default):

PLAY [myservers] ***************************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [192.168.1.3]

TASK [User Creation] ***********************************************************************************
changed: [192.168.1.3]

PLAY RECAP *********************************************************************************************
192.168.1.3                : ok=2    changed=1    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

[testuser@host2 ~]$
```

Manage variables and facts

Lab exercise create username/passwd file and playbook both encrypted.

```
[testuser@host2 ~]$ ansible-vault view passwd_2.yml
Vault password:
---
 user_name: test007
 password:
$6$TA30OBFfXhocCtJz$8lhMfQ2JADNBKbe7ENteawr5tde7onM2NnAwuXKMmtsCzvpUmhwPzysjRzNWUfH3vp
f.uml3eWzEE.X0u7cki/
...
[testuser@host2 ~]$
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-vault view plybook_2.yml
Vault password:
---
 -
   hosts: myservers
   vars_files: passwd_2.yml
   tasks:
        - name: User Creation with password.
          user:
                name: "{{user_name}}"
                password: "{{password}}"
                comment: ANsibleUser
                state: present
...
```

Implementing Task Control

Agenda:

- **Implement loops to write efficient tasks.**
- **register variable**
- **Running tasks conditionally**
- **task handlers**
- **Handling Task Failure**

Implementing Task Control

Task Iteration with Loops:

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

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

Implementing Task Control

Task Iteration with Loops:

```
[testuser@host2 myplaybooks]$ cat loop_1.yml
---
-
 hosts: myservers
 tasks:
        - name: Run commands
          shell: "{{ item }}"
          loop:
                - "touch one.txt"
                - "ls –lrt > log.txt"
...
[testuser@host2 myplaybooks]$
```

Implementing Task Control

Register variables with Loops:

Ansible registers are used when user want to capture the output of a task to a variable. We can then use the value of these registers for different scenarios like a conditional statement, logging etc.

Implementing Task Control

Register variables with Loops:

```
[testuser@host2 ~]$ cat loop_register.yml
---
-
  hosts: myservers
  tasks:
        - name: Running commands over managed host
          shell: "{{item}}"
          loop:
              - "ls -lrt /tmp"
              - "ls -lrt /home"
          register: info

        - name: Capturing the output of the commands
          debug: var=info
...
[testuser@host2 ~]$
```

Implementing Task Control

Running tasks Conditionally (when):

The ***when*** statement is used to run a task conditionally. It takes as a value the condition to test. If the condition is met, the task runs. If the condition is not met, the task is skipped.

One of the simplest conditions that can be tested is whether a Boolean variable is true or false.

Implementing Task Control

Running tasks Conditionally (when):

```
[testuser@host2 ~]$ cat condition1.yml
---
-
 hosts: myservers
 vars:
        package: httpd
        condition: true
 tasks:
        - name: Installing "{{package}}" package
          yum:
                name: httpd
                state: latest
          when: condition

...
[testuser@host2 ~]$
```

Implementing Task Control

Running tasks Conditionally (when):

```
[testuser@host2 ~]$ cat condition2.yml
---
-
 hosts: myservers
 vars:
        condition: true
        package: httpd      #comment this line to undefine the package variable
 tasks:
        - name: Installing "{{package}}" package
          yum:
                name: "{{package}}"
                state: latest
          when: package is defined

...
```

Implementing Task Control

Running tasks Conditionally (using *when* with ansible facts/multiple conditions):

```
[testuser@host2 ~]$ cat condition4.yml
---
-
 hosts: myservers
 tasks:
        - name: Installing the HTTPD package.
          yum:
                name: httpd
                state: latest
          when: >
                (ansible_distribution == "RedHat"  and ansible_distribution_version == "8.1")
                or
                (ansible_kernel == "4.18.0-221.el8.x86_64")
...
[testuser@host2 ~]$
```

Implementing Task Control

Running tasks Conditionally (using *in* keyword)

Used when Comparing a variable against a list

Implementing Task Control

Running tasks Conditionally (using in keyword)

```
[testuser@host2 ~]$ cat  ./condition_in.yml
---
-
 hosts: myservers
 vars:
        package: postfix
        osDistros:
                - RedHat
                - Fedora
                - CentOS
 tasks:
        - name: Install "{{package}}" package
          yum:
                name: "{{package}}"
                state: latest
          when: ansible_distribution in osDistros
…
```

Implementing Task Control

Running tasks Conditionally (combining loops and conditional tasks)

```
[testuser@host2 ~]$ cat  ./condition6.yml
---
-
   hosts: myservers
   tasks:
         - name: Installing postfix package
           yum:
                 name: postfix
                 state: latest

           loop: "{{ansible_mounts}}"
           when: item.mount == "/" and item.size_available > 4*1024*1024*1024
...
[testuser@host2 ~]$
```

Implementing Task Control

Handlers

Sometimes you want a task to run only when a **change** is made on a machine. For example, you may want to restart a service if a task updates the configuration of that service, but not if the configuration is unchanged. Ansible uses handlers to address this use case. **Handlers** are tasks that only run when **notified**. Each handler should have a globally unique name.

Handlers can be considered as inactive tasks that only get triggered when explicitly invoked using a ***notify*** statement.
The ***handlers*** keyword indicates the start of the list of handler tasks.

If a task that includes a **notify** statement does not report a changed result (for example, a package is already installed and the task reports ok), the handler is not notified. The handler is skipped unless another task notifies it. Ansible notifies handlers only if the task reports the changed status.

Implementing Task Control

Handlers

```
[testuser@host2 ~]$ cat handlers.yml
---
-
  hosts: myservers
  vars:
        package_name: postfix
  tasks:
        - name: Installing "{{package_name}}"
          yum:
                name: "{{package_name}}"
                state: latest
          notify:
             - restart_service

  handlers:
        - name: restart_service
          service:
                name: postfix
                state: restarted

...
```

Implementing Task Control

Handling Task Failures

Ansible evaluates the return code of each task to determine whether the task succeeded or failed. Normally, when a task fails Ansible immediately aborts the rest of the play on that host, skipping all subsequent tasks.

However, sometimes you might want to have play execution continue even if a task fails. For example, you might expect that a particular task could fail, and you might want to recover by running some other task conditionally. There are a number of Ansible features that can be used to manage task errors.

Implementing Task Control

Handling Task Failures

Ignoring Task Failure

By default, if a task fails, the play is aborted. However, this behavior can be overridden by ignoring failed tasks. You can use the ***ignore_errors*** keyword in a task to accomplish this.

having ***ignore_errors*** set to ***yes*** allows execution to continue.

Implementing Task Control

Handling Task Failures (*Ignoring Task Failure)*

```
[testuser@host2 ~]$ cat  ./task_failure_1.yml
---
-
 hosts: myservers
 tasks:
        - name: Start an imaginary service
          service:
                name: nomads
                state: restarted
          ignore_errors: yes
        - name: Copy some files to managed hosts
          copy:
            src: ./handlers.yml
            dest: /tmp

…
```

Implementing Task Control

Handling Task Failures

Specifying Task Failure Conditions

You can use the ***failed_when*** keyword on a task to specify which conditions indicate that the task has failed.
This is often used with command modules that may successfully execute a command, but the command's output indicates a failure.

Implementing Task Control

Handling Task Failures (*Specifying Task Failure Conditions*)

```
---
-
 hosts: myservers
 tasks:
        - name: Start an imaginary service
          service:
                name: nomads
                state: restarted
          ignore_errors: yes
        - name: Copy some files to managed hosts
          copy:
            src: ./myscript.sh
            dest: /tmp
        - name: Running a shell script
          shell: bash /tmp/myscript.sh
          register: information
          failed_when: "'script failed' in information.stdout"
        - name: Installing postfix package
          yum:
                name: postfix
                state: latest
...
```

Deploying Files to Managed Hosts

Agenda:

- Create, edit, and remove files on managed hosts.
- manage permissions, ownership,
- Customize files using Jinja2 templates, and deploy them to managed hosts

Deploying Files to Managed Hosts

Red Hat Ansible Engine provides collection of modules.
To make it easier to organize, document, and manage them, they are organized into groups based on function in the documentation and when installed on a system.

The Files modules library includes modules allowing you to accomplish most tasks related to Linux file management, such as creating, copying, editing, and modifying permissions and other attributes of files.

https://docs.ansible.com/ansible/2.9/modules/list_of_files_modules.html

Deploying Files to Managed Hosts

Commonly Used Files Modules

MODULE NAME	MODULE DESCRIPTION
`blockinfile`	Insert, update, or remove a block of multiline text surrounded by customizable marker lines.
`copy`	Copy a file from the local or remote machine to a location on a managed host. Similar to the `file` module, the `copy` module can also set file attributes, including SELinux context.
`fetch`	This module works like the `copy` module, but in reverse. This module is used for fetching files from remote machines to the control node and storing them in a file tree, organized by host name.
`file`	Set attributes such as permissions, ownership, SELinux contexts, and time stamps of regular files, symlinks, hard links, and directories. This module can also create or remove regular files, symlinks, hard links, and directories. A number of other file-related modules support the same options to set attributes as the `file` module, including the `copy` module.
`lineinfile`	Ensure that a particular line is in a file, or replace an existing line using a back-reference regular expression. This module is primarily useful when you want to change a single line in a file.
`stat`	Retrieve status information for a file, similar to the Linux **`stat`** command.

Deploying Files to Managed Hosts

file module (Manage files and file properties)

- File creation,
- Set attributes of files, symlinks or directories.
- Alternatively, remove files, symlinks or directories.

Deploying Files to Managed Hosts

file module (Manage files and file properties)

```
[testuser@host2 ~]$ cat file_1.yml
---
-
  hosts: myservers
  tasks:
        - name: Creating a file
          file:
                path: /tmp/file_creation.txt
                mode: 0644
                owner: testuser
                group: testuser
                state: touch
                #state directory
...
[testuser@host2 ~]$
```

Deploying Files to Managed Hosts

file module (Manage files and file properties)

```
[testuser@host2 ~]$ cat file_1.yml
---
-
  hosts: myservers
  tasks:
        - name: Deleting a file
          file:
                path: /tmp/handlers.yml
                state: absent
...
[testuser@host2 ~]$
```

Deploying Files to Managed Hosts

copy module (Manage files and file properties)

```
[testuser@host2 ~]$ cat copy.yml
---
-
  hosts: myservers
  tasks:
        - name: Copying files to the managed hosts
          copy:
            src: ./myscript.sh
            dest: /tmp
            force: yes
...
[testuser@host2 ~]$
```

Deploying Files to Managed Hosts

fetch module (Copy files from managed hosts)

```
[testuser@host2 ~]$ cat fetch.yml
---
-
  hosts: myservers
  tasks:
        - name: Copying files from managed hosts to control server
          fetch:
              src: /tmp/from_managed_hosts.txt
              dest: /tmp
...
[testuser@host2 ~]$
```

Deploying Files to Managed Hosts

lineinfile module

```
[testuser@host2 ~]$ cat lineinfile.yml
---
-
  hosts: myservers
  tasks:
        - name: Copying files to the managed hosts
          copy:
            src: ./file.txt
            dest: /tmp
            force: yes
        - name: Add line in file.
          lineinfile:
                path: /tmp/file.txt
                line: I am adding a new line xxx
                state: present
...
[testuser@host2 ~]$
```

Deploying Files to Managed Hosts

blockinfile module ***(***The block will be surrounded by a marker***)***

```
---
-
  hosts: myservers
  tasks:
        - name: Copying files to the managed hosts
          copy:
            src: ./file.txt
            dest: /tmp
            force: yes
        - name: Add line in file.
          blockinfile:
                path: /tmp/file.txt
                block: |
                      I am adding this new block
                      This is a new block to show the test
                      This is the third line in this block
                state: present
...
```

Deploying Files to Managed Hosts

stat module

```
[testuser@host2 ~]$ cat stat.yml
---
-
  hosts: myservers
  tasks:
        - name: Getting stats of a file.
          stat:
                path: /root/rhel-8.3-beta-1-x86_64-dvd.iso
          register: stats_output
        - name: Disply the file stats
          debug:
            var: stats_output
...
[testuser@host2 ~]$ stat -c=%X,%Y,%Z,%b file_name
```

Deploying Files to Managed Hosts

Introduction to Jinja2

Template:

A template contains variables which are replaced by the values which are passed in when the template is *rendered.* Variables are helpful with the dynamic data.

Templating is powerful way to modify the files.
With this method, you can write a template configuration file that is automatically customized for the managed host when the file is deployed, using Ansible variables and facts.

This can be easier to control and is less error-prone.

Deploying Files to Managed Hosts

Introduction to Jinja2

Ansible uses the Jinja2 templating system for template files.

Ansible also uses Jinja2 syntax to reference variables in playbooks, so you already know a little bit about how to use it.

Deploying Files to Managed Hosts

Deploying Jinja2 templates

Jinja2 templates are a powerful tool to customize configuration files to be deployed on the managed hosts.

When the Jinja2 template for a configuration file has been created, it can be deployed to the managed hosts using the ***template*** module, which supports the transfer of a local file on the control node to the managed hosts.

Deploying Files to Managed Hosts

Introduction to Jinja2

{{ EXPR }} are used for outputting the results of an expression or a variable to the end user.

{% EXPR %} for expressions or logic (for example if condition, loops)

{# COMMENT #} syntax to enclose comments that should not appear in the final file.

Deploying Files to Managed Hosts

Introduction to Jinja2 (Basic jinja2 template)

[testuser@host2 ~]$ cat jinja1.j2

```
{# This is file to configure the network #}
This is the first line of the {{file_name}} file
*******************************************
system_distro: {{ansible_facts['distribution']}}
{% if rhel8 %}
system_distro_version: RHEL8
{% endif %}
system_name: {{ansible_facts['hostname']}}
system_IP: {{ansible_facts['default_ipv4']['address']}}
system_MAC: {{ansible_facts['default_ipv4']['macaddress']}}
Permit Root Login: {{permit_root_login}}
Allowed Users: {{allowed_usears}}
*******************************************
```

Deploying Files to Managed Hosts

Introduction to Jinja2 (Basic jinja2 template in playbook)

```
[testuser@host2 ~]$ cat jinja1.yml
---
 -
   hosts: myservers
   vars:
         file_name: network.cfg
         permit_root_login: no
         allowed_usears: All
         rhel8: true

   tasks:
         - name: Creating a template file.
           template:
                 src: jinja1.j2
                 dest: /tmp/network.cfg
...
[testuser@host2 ~]$
```

Deploying Files to Managed Hosts

Introduction to Jinja2 (Basic jinja2 template rendered o/p)

[root@RHEL-83 tmp]# cat network.cfg

This is the first line of the network.cfg file
****
system_distro: RedHat
system_distro_version: RHEL8
system_name: RHEL-83
system_IP: 192.168.1.3
system_MAC: 08:00:27:bd:d9:d5
Permit Root Login: False
Allowed Users: All
****

Managing Large Projects

Agenda:

- **Configure Parallelism in Ansible Using Forks**
- **Managing Rolling Updates**

Managing Large Projects

Configure Parallelism in Ansible Using Forks

When Ansible processes a playbook, it runs each play in order. After determining the list of hosts for the play, Ansible runs through each task in order.

In theory, Ansible could simultaneously connect to all hosts in the play for each task. This works fine for small lists of hosts. But, if the play targets hundreds of hosts, this can put a heavy load on the control node.

Managing Large Projects

Configure Parallelism in Ansible Using Forks

The maximum number of simultaneous connections that Ansible makes is controlled by the forks parameter in the Ansible configuration file.
It is set to 5 by default

```
#local_tmp      = ~/.ansible/tmp
#plugin_filters_cfg = /etc/ansible/plugin_filters.yml
#forks          = 5
#poll_interval  = 15
```

```
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-config dump | grep -i forks
DEFAULT_FORKS(default) = 5
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-config list | grep -i forks
DEFAULT_FORKS:
  description: Maximum number of forks Ansible will use to execute tasks on target
  - {name: ANSIBLE_FORKS}
  - {key: forks, section: defaults}
  name: Number of task forks
```

Managing Large Projects

Configure Parallelism in Ansible Using Forks

User can override the default setting for forks from the command line in the Ansible configuration file.

Both the ansible and the ansible-playbook commands offer the ***-f or --forks*** options to specify the number of forks to use.

Managing Large Projects

Managing Rolling Updates

Normally, when Ansible runs a play, it makes sure that all managed hosts have completed each task before starting the next task.

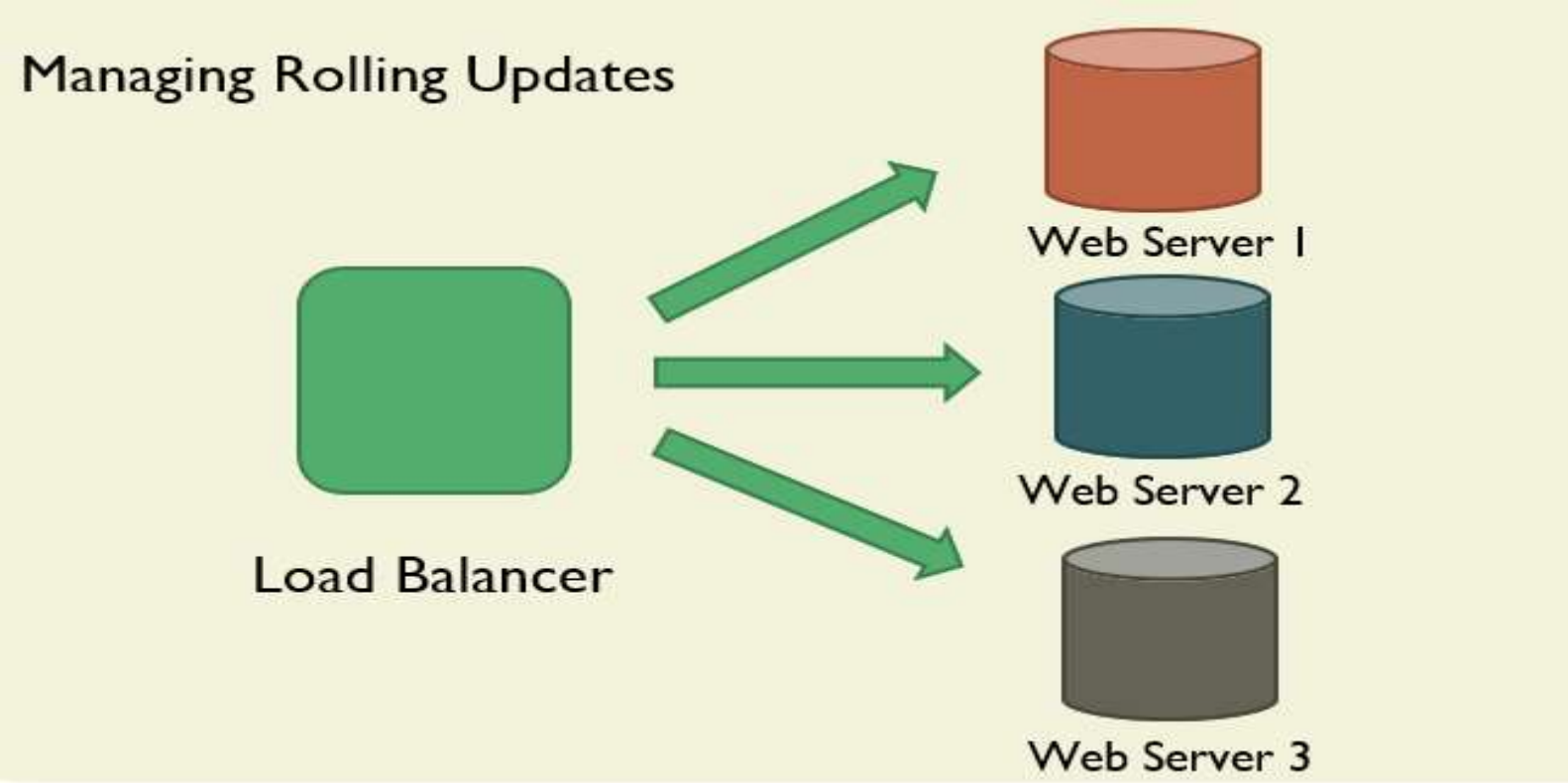
After all managed hosts have completed all tasks, then any notified handlers are run.

However, running all tasks on all hosts can lead to undesirable behavior.

For example, if a play updates a cluster of load balanced web servers, it might need to take each web server out of service while the update takes place. If all the servers are updated in the same play, they could all be out of service at the same time.

Managing Large Projects

Managing Rolling Updates



Managing Large Projects

Managing Rolling Updates

One way to avoid this problem is to use the ***serial*** keyword to run the hosts through the play in batches.

Each batch of hosts will be run through the entire play before the next batch is started. In the example below, Ansible executes the play on two managed hosts at a time, until all managed.

```
---
- name: Rolling update
  hosts: webservers
  serial: 2
  tasks:
  - name: latest apache httpd package is installed
    yum:
      name: httpd
      state: latest
    notify: restart apache

  handlers:
  - name: restart apache
    service:
      name: httpd
      state: restarted
```

Managing Large Projects

Managing Rolling Updates

```
---
- name: test play
  hosts: webservers
  serial:
    - 1
    - 5
    - 10
```

In the above example, the first batch would contain a single host, the next would contain 5 hosts, and (if there are any hosts left), every following batch would contain either 10 hosts or all the remaining hosts, if fewer than 10 hosts remained.

Ansible Roles

Agenda:

- Defining Ansible roles
- Advantages of Ansible roles
- How to create ansible roles
- Ansible Role directory structure
- Lab session

Ansible Roles

Ansible Roles

Ansible role is the primary mechanism for breaking a playbook into multiple files.

This ***simplifies*** writing complex playbooks, and it makes them easier to reuse.

The breaking of playbook allows you to logically break the playbook into reusable components.

Ansible Roles

Advantages of Ansible roles:

- Roles group content, allowing easy sharing of code with others.
- Roles can be written that define the essential elements of a system type: web server, database server, Git repository, or other purpose
- Roles make larger projects more manageable.
- Roles can be developed in parallel by different administrators

Ansible Roles

Creating Ansible roles (ansible-galaxy command)

```
[testuser@host2 nbs_roles]$ ansible-galaxy init /home/testuser/nbs_roles/webserver --offline
- Role /home/testuser/nbs_roles/webserver was created successfully
[testuser@host2 nbs_roles]$
[testuser@host2 nbs_roles]$ tree webserver
webserver
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml

8 directories, 8 files
[testuser@host2 nbs_roles]$
```

Ansible Roles

Subdirectory	Function
defaults	The **main.yml** file in this directory contains the default values of role variables that can be overwritten when the role is used. These variables have low precedence and are intended to be changed and customized in plays.
files	This directory contains static files that are referenced by role tasks.
handlers	The **main.yml** file in this directory contains the role's handler definitions.
meta	The **main.yml** file in this directory contains information about the role, including author, license, platforms, and optional role dependencies.
tasks	The **main.yml** file in this directory contains the role's task definitions.
templates	This directory contains Jinja2 templates that are referenced by role tasks.
tests	This directory can contain an inventory and **test.yml** playbook that can be used to test the role.
vars	The **main.yml** file in this directory defines the role's variable values. Often these variables are used for internal purposes within the role. These variables have high precedence, and are not intended to be changed when used in a playbook.

Ansible Roles

Lab session

Troubleshoot Ansible

Agenda:

- **Troubleshoot generic issues while creating playbook and repair them.**
- **Troubleshoot failures on managed hosts when running a playbook.**

Troubleshoot Ansible

- Log files for Ansible
- The Debug Module
- Managing Errors

Troubleshoot Ansible

Log files for Ansible

By default, Red Hat Ansible Engine is not configured to log its output to any log file. We can enable built-in logging infrastructure by configuring

- **log_path** parameter in ansible.cfg configuration file.
- Setting **$ANSIBLE_LOG_PATH** environment variable.

```
# logging is off by default unless this path is defined
# if so defined, consider logrotate
#log_path = /var/log/ansible.log
```

Troubleshoot Ansible

Debug Module

The debug module provides insight into what is happening in the play. This module can display the value for a certain variable at a certain point in the play. This feature is key to debugging tasks that use variables to communicate with each other.

> DEBUG (/usr/lib/python3.6/site-packages/ansible/modules/utilities/logic/debug.py)

Debug module prints statements during execution and can be useful for debugging variables or expressions without necessarily halting the playbook.

Troubleshoot Ansible

Managing Errors

There are several issues than can occur during a playbook run, mainly related to the syntax of either the playbook or any of the templates it uses, or due to connectivity issues with the managed clients. Those errors are issued by the ansible-playbook command at execution time.

--syntax-check option, which checks the YAML syntax for the playbook.

--step command interactively prompts for confirmation that you want each task to execute.

--start-at-task option allows you to start execution of a playbook from a specific task.

Troubleshoot Ansible

Managing Errors

```
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-playbook -i ./hosts_test --syntax-check playbook_1.yaml

playbook: playbook_1.yaml
[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-playbook -i ./hosts_test playbook_1.yaml --step

PLAY [This is my test Playbook] ************************************************************
Perform task: TASK: Gathering Facts (N)o/(y)es/(c)ontinue: y

Perform task: TASK: Gathering Facts (N)o/(y)es/(c)ontinue: ********************************************

[testuser@host2 ~]$
[testuser@host2 ~]$ ansible-playbook -i ./hosts_test playbook_1.yaml  --start-at-task="start httpd service"
```

Troubleshoot Ansible

Listing tasks

```
[testuser@host2 ~]$ ansible-playbook -i ./hosts_test playbook.yaml  --list-tasks

playbook: playbook.yaml

  play #1 (myservers): This is my first Play Using Variables    TAGS: []
    tasks:
      First task check dirs present in home     TAGS: []
      This is my Second task to print the output        TAGS: []
      This is my Third task to create the user  TAGS: []
      Fourth task check dirs present in home    TAGS: []
      This is my Final task to print the output TAGS: []

  play #2 (myservers): This is my second Play to enable HTTP service Using Variables    TAGS: []
    tasks:
      First Install the httpd package   TAGS: []
      Now Create the default index.html TAGS: []
      Now enbale HTTP service in firewall       TAGS: []
      Finally start  HTTP service       TAGS: []
[testuser@host2 ~]$
```

Troubleshoot Ansible

Verbosity Configuration

verbosity of the output from ansible-playbook can be increased by adding one or more -v options.

The ansible-playbook -v command provides additional debugging information, with up to four total levels.

Option	Description
-v	The output data is displayed.
-vv	Both the output and input data are displayed.
-vvv	Includes information about connections to managed hosts.
-vvvv	Includes additional information such scripts that are executed on each remote host, and the user that is executing each script.


Thank you