Ansible Fundamentals

Case Study - 1

You need to manage a user, .

You care specifically about:

- his existence
- his primary group
- his home directory

Case Study - 1

- Tools built into most distro’s that can help:

- useradd
- usermod
- groupadd
- groupmod
- mkdir
- chmod
- chgrp
- chown

Case Study - 1

- Platform idiosyncrasies:
  - Does this box have ‘useradd’ or ‘adduser’?
  - What was that flag again?
  - What is difference between ‘–l’ and ‘–L’?
  - What does ‘–r’ means
  - Recursive
  - Remove read privileges
  - System user

- If I run this command again, what will it do?

Case Study - 1

- You could do something like this:

```
#!/bin/sh
USER=$1 ; GROUP=$2 ; HOME=$3
if [ 0 –ne $(getent passwd $USER > /dev/null)$? ]
then useradd $USER –home $HOME –gid $GROUP –n ; fi
OLDGID=`getent passwd $USER | awk –F: '{print $4}'`
OLDGROUP=`getent group $OLDGID | awk –F: '{print $1}'`
OLDHOME=`getent passwd $USER | awk –F: '{print $6}'`
if [ "$GROUP" != "$OLDGID" ] && ["$GROUP" != "$OLDGROUP" ]
then usermod –gid $GROUP $USER; fi
if [ "$HOME" != "$OLDHOME" ]
then usermod –home $HOME $USER; fi
```

Case Study - 1

What About?

- Robust error checking?
- Solaris and Windows support?
- Robust logging of changes?
- Readable code?
- What if need to create in 1000+ Servers?

Case Study – 1

Ansible Way of Configuration Management:

```
tasks:
 - name: Creating sandeep User
    user: name=Sandeep comment=“sandeep Singh"  state=present

tasks:
 - name: Creating Singh Group
    group: name=singh state=present
```

Ansible Way: Maintaining State

- You(Even Ansible can do it on cloud) provision a node.
- Ansible configures it.
- Ansible maintains the desired state when needed.

Note: You make to sure the state is configured as per environment requirements.

Ansible : Infrastructure as Code

- Descriptive
- Straightforward
- Transparent

```
[root@sandeep]# cat ntp.yml
---
# This is my Host section
- hosts: localhost
# This is my Task section
  tasks:
  - name: NTP Installation
    yum: name=ntp state=present
  - name: NTP Service
    service: name=ntpd state=started enabled=yes
```

Ansible : Idempotency

- Ansible enforces in an idempotent way.
- The property of certain operations in mathematics or computer science is that they can be applied multiple times without further changing the result beyond the initial application.
- Able to be applied multiple times with the same outcome.

Ansible Terminology

- **Controller/Master:** The Ansible master is the machine that controls the infrastructure and dictates policies for the servers it manages. It operates both as a repository for configuration data and as the control center that initiates remote commands and ensures the state of your other machines.

- **Managed/Agent Nodes :** The servers that Ansible configures are called Clients/Nodes.

- **Ansible Inventory:** Ansible Inventory represents which machines it should manage using a very simple INI file that puts all of your managed machines in groups of your own choosing.

- **Ansible Adhoc-tasks:** Ansible uses adhoc requests to confirm simple and small tasks on any server right a way without login into the client. The best example is to check the Alive Status for whole managed inventory.

Ansible Terminology

Playbooks: A structured way to put all of the defines tasks for your application or your whole setup.

Modules: In built functions which executes at the backend to perform underlined tasks in Ansible.

***yml files:** describe a set of desired states that a system needs to be in, for example “apache needs to be installed and running”.

Ansible Tower: Ansible Tower by Red Hat helps is a web-based solution that makes Ansible even more easy to use for IT teams of all kinds. It’s designed to be the hub for all of your automation tasks.

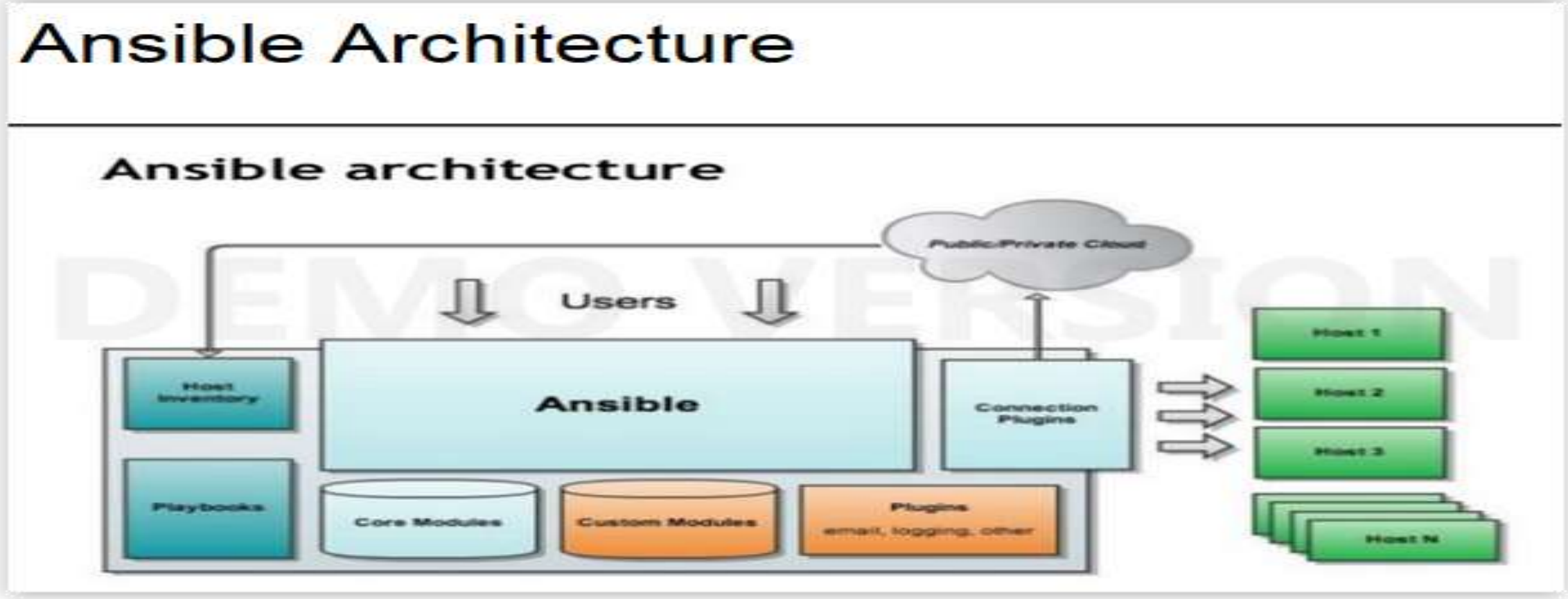
Ansible Terminology

- **Ansible Galaxy:** Ansible Galaxy is a free site for finding, downloading, and sharing community developed roles. Downloading roles from Galaxy is a great way to jumpstart your automation projects..

- **Ansible for Unix/Linux:** The Ansible master communicates and manage Unix/Linux Clients using SSH by default.

- **Ansible for Windows:** Starting in version 1.7, Ansible also contains support for managing Windows machines. This uses native PowerShell remoting, rather than SSH. and uses the “winrm” Python module to talk to remote hosts.

Ansible Communications



Ansible Communication test

- Communication checks with password authentications:

```
[root@controller ~]# ansible centos-managed -m ping --ask-pass
SSH password:
centos-managed | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Ansible Modules

- Modules are the Basic Building Block of Ansible.
- These are the readymade tools to perform various tasks and operations on “Managed Nodes”.
- Modules can be used with Ansible Ad-Hoc Remote Executions and/or Playbooks as a core building blocks.
- Ansible Ships with multiple in-built Modules (approx. 2000+ in Ansible 2.7).
- Can be used for Standalone servers, Virtual Machines and for any Public/Private Cloud Instances.

Ansible Modules

- Two types of Ansible Modules: Core Modules & Custom Modules.
- Robust Module Documentation on website.
- Command line utility on Module information and usage.
- CLI utility ansible-doc on “Controller Node”.
- Execute ansible-doc -l to list all available Modules.
- Execute ansible-doc <module-name> to find all details about Modules.
- For GUI refer **“http://docs.ansible.com/ansible/modules_by_category.html”**

Ansible Adhoc Execution

- Easy to learn ad-hoc command line utility - “ansible”.
- Quick On-demand tasks on “Managed Nodes”.
- 1 to 1 approach, single ad-hoc command is used to perform single operation.
- Multiple ad-hoc operations require multiple “ansible” ad-hoc run.
- Ad-hoc execution syntax.
- Ansible Ping Communication Test with “Managed Nodes”.
- Various real time examples with ad-hoc execution.

Ansible Adhoc Execution

- Let’s try executing a remote command, before that make sure you have out an entry of the host in “/etc/ansible/hosts”

- Connect to the master and type:

ansible <host-name/IP> -m ping --ask-pass

ansible "*" -m ping --ask-pass

- First argument = target client
- Second argument = function to execute
- Other arguments = params for the function

Ansible Adhoc Execution

Adding Username and Connection method in “/etc/ansible/hosts”:

You can specify each host for specific connection type/port and connection username:

- ansible_host : The name of the host to connect to, if different from the alias you wish to give
- ansible_port: The ssh port number, if not 22
- ansible_user: The default ssh user name to use.
- ansible_ssh_pass: The ssh password to use (never store this variable in plain text; always use a vault. See Variables and Vaults)
- ansible_ssh_private_key_file: Private key file used by ssh. Useful if using multiple keys and you don’t want to use SSH agent.

user20-client ansible_connection=ssh ansible_user=centos

Ansible Adhoc Execution

- There are a bunch of predefined :

«**Ad-Hocmodules**»

«**execution modules**»

«**Ad-Hocmodules**»: ansible all/"Client or Group“ -a “<adhoc-command>" --ask-pass

«**execution modules**» ansible all/"Client or Group“ -m <module_name> -a <arguments>

- Note: To list all Ansible Modules run below command:

ansible-doc -l

Ansible Adhoc Execution

- For example, executing a shell commands:

ansible 192.168.74.51 -a "ls -l /tmp" --ask-pass

ansible 192.168.74.51 -a "uname -a" --ask-pass

ansible 192.168.74.51 -a "cat /etc/redhat-release" --ask-pass

ansible 192.168.74.51 -a 'service ntpd status' --ask-pass

ansible “*” --list-hosts // Its’ll show all the hosts that’ll effect with the command

Ansible Adhoc Execution

- For example, executing Ansible Modules:

ansible 192.168.74.51 -m ping --ask-pass

ansible 192.168.74.51 -m user -a "name=sandeep state=present" --ask-pass

ansible 192.168.74.51 -m file -a "path=/var/tmp/sandeep mode=777 group=sandeep state=touch" --ask-pass

ansible 192.168.74.51 -m service -a "name=ntpd state=stopped" --ask-pass

ansible 192.168.74.51 -m file -a "path=/var/yog/rah/test mode=777 state=directory" --ask-pass

Ansible Adhoc : Lab

- Check the uptime using Ansible Ad-hoc execution with password-less authentication
- Check OS release using Ansible Ad-hoc execution with password-less authentication
- Install a package named “telnet” on managed host
- Create a user named “yourname” with bash shell having user id of 9999 on managed host
- Create a file named “/tmp/myfile” with all permissions and user and owner root on managed host
- Copy a local file to remote machine
- Run multiple commands parallelly with shell module.

Facts

Ansible uses “facts” to gather information about the host system (any host).

“ansible <client-name> -m setup”

“ansible <client-name> -m setup –ask-pass”

Command returns a list of key value pairs (specific to Ansible).

The returned key value pairs are “facts”. Example:

- [root@user20-master ~]# ansible user20-client -m setup | grep -i ansible_user_id
- "ansible_user_id": "centos",
- [root@user20-master ~]# ansible user20-client -m setup -b | grep -i ansible_user_id
- "ansible_user_id": "root",
- [root@user20-master ~]#

Ansible Playbooks

Ansible Playbooks

- YAML Structure
- Ansible Playbooks
- Playbooks Structure
- Playbooks Syntax
- Playbooks Pre-Execution
- Playbooks Smoke Test
- Playbooks Real Time Run

YAML Structure

Though YAML syntax may seem daunting and terse at first, there are only three very simple rules to remember when writing YAML for Playbooks.

- Rule One: **Indentation**
  - YAML uses a fixed indentation scheme to represent relationships between data layers. Ansible requires that the indentation for each level consists of exactly two spaces. **Do not use tabs.**

YAML Structure

Rule Two: **Colons**

- Dictionary keys are represented in YAML as strings terminated by a trailing colon. Values are represented by either a string following the colon, separated by a space:

```
my_key: my_value
```

- In Python, the above maps to:

```
{'my_key': 'my_value'}
```

- Alternatively, a value can be associated with a key through indentation.

```
my_key:
  my_value
```

YAML Structure

Rule Three: **Dashes**

- To represent lists of items, a single dash followed by a space is used. Multiple items are a part of the same list as a function of their having the same level of indentation.

```
- list_value_one
- list_value_two
- list_value_three
```

- Lists can be the value of a key-value pair. This is quite common in Ansible:

```
my_dictionary:
  - list_value_one
  - list_value_two
  - list_value_three
```

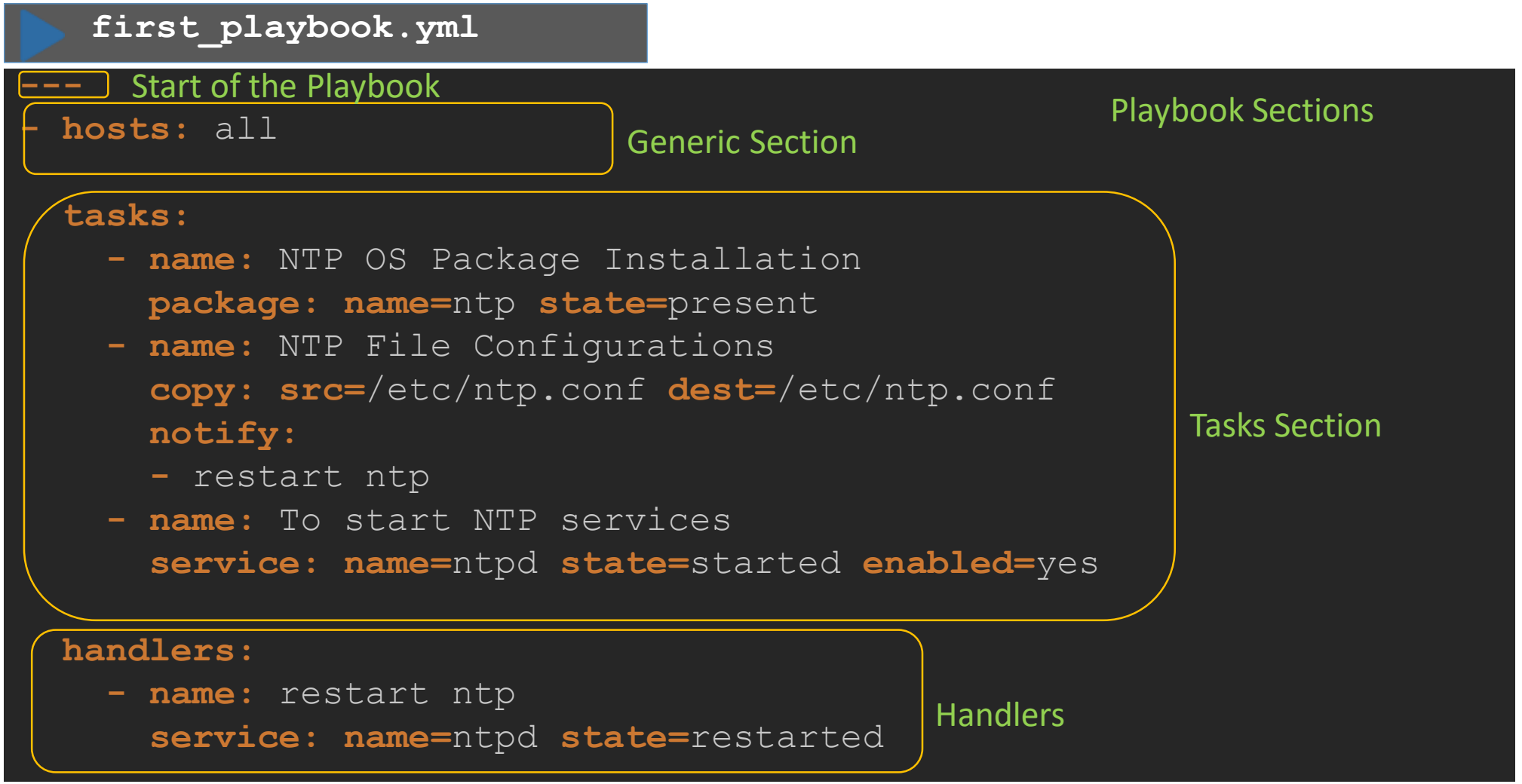
Ansible Playbooks

- **Ansible Playbooks are the file in YAML format with sequential instructions to perform operations on Managed Nodes.**
- Written in YAML (**YAML** Ain't Markup Language).
- In simple layman language Playbooks are simple YAML files containing implementation steps.
- Opposite to Ad-Hoc requests, no restrictions on running multiple operations on managed nodes in a single run.
- Very easy to write and understand than other configuration Management tools.
- Ansible Playbooks file extension is .yml or yaml.

Ansible Playbooks

- The Playbook are typically divided into three parts, with YAML format:
  - Start of a Play (Hosts, variables, connections and Users)
  - Tasks list
  - Handlers

Ansible Playbook



Playbooks Structure

- Playbooks in Ansible are "collection of Plays".
- **Plays** are nothing but a “collection of attributes“ and a “sequence of operations” to be performed on Managed Nodes.
- “Collection of Attributes” defines set of Managed Nodes, Communication Connection, Privilege escalations and different variable associated with playbooks.
- The "sequence of operations" is called "**Tasks**" in Ansible.
- Every Play is associated with one or more Ansible "Tasks".

Playbooks Structure

- The Tasks are created with the help of Ansible Modules to perform actions on Managed Nodes.
- Plays are associated with Managed Nodes and are called "hosts".
- Plays are associated with "Managed Nodes" to perform action on them.
- Playbooks are executed on Managed Nodes in sequence of contents written inside, so order of contents inside Playbook matters.
- Playbooks may have multiple plays associated with set of different "Managed Nodes".

Playbooks Structure

- **Start of a Play - Basic (Hosts and Users):**
- First part of Ansible Playbooks
- Provides the hosts/group of hosts to target
- Provides which user you want to perform the tasks
- For each play in a playbook, you get to choose which machines in your infrastructure to target and what remote user to complete the steps (called tasks)

Examples:

```
---
- hosts: webservers
  remote_user: root
```

Playbooks Structure

Examples:

You can also login as you, and then become a user different than root:

```
---
- hosts: webservers
  remote_user: yourname
  become: yes
  become_user: postgres
```

You can also use other privilege escalation methods, like su:

```
---
- hosts: webservers
  remote_user: yourname
  become: yes
  become_method: su
```

Playbooks Structure

- **Tasks list:**

- Each play contains a list of tasks. Tasks are executed in order, one at a time, against all machines matched by the host pattern, before moving on to the next task. When running the playbook, which runs top to bottom, hosts with failed tasks are taken out of the rotation for the entire playbook. If things fail, simply correct the playbook file and rerun.

- The goal of each task is to execute a module, with very specific arguments

Playbooks Structure

Here is what a basic task looks like. As with most modules, the service module takes key=value arguments:

```
tasks:
  - name: make sure apache is running
    service: name=httpd state=started
```

Variables can be used in action lines. Suppose you defined a variable called vhost in the vars section, you could do this:

```
tasks:
  - name: create a virtual host file for {{ vhost }}
    template: src=somefile.j2 dest=/etc/httpd/conf.d/{{ vhost }}
```

Playbooks Structure

- **Handlers:**

- In simple layman language Handlers are **Running Operations On Changes.**

- Handlers are lists of tasks, not really any different from regular tasks, that are referenced by a globally unique name, and are notified by notifiers.

- If nothing notifies a handler, it will not run.

- Regardless of how many tasks notify a handler, it will run only once, after all of the tasks complete in a particular play.

- The things listed in the notify section of a task are called handlers.

Playbooks Syntax

- Let's do some Ansible Playbooks creations:

- Creation of user
- Creation of group
- Package Installation/Service Operations
- Files and Directories

Playbooks Syntax

- Creation of user:

```
[root@sandeep]#cat user.yml
---
- hosts: all
  tasks:
   - name: Creating user
     user:
      name: sandeep
      state: present
   - name: Creating another user
     user: name=sumit state=present shell=/bin/sh
```

Playbooks Syntax

- Creation of group:

```
[root@ansible]#cat user.yml
---
- hosts: all
  tasks:
    - name: Group Creation
      group:
       name: group
       gid:  5555
    - name: Another Group Creation
      group: name=grp2 gid=5656
```

Thanks You

Demo

Playbooks Run

- Perform syntax check.
- Perform Dry Run Test.
- Perform Real time run.
- Check Results.

Playbooks Pre-execution

- Before running the Playbooks, lets explore some useful tips:
- Finding Modules and Attributes:
- ansible-doc –l
- ansible-doc -v service

- To see what hosts would be affected by a playbook before you run it:
- ansible-playbook playbook.yml --list-hosts

- Command to see all FACTS:
- ansible all -m setup --ask-pass

Playbooks Syntax Checks

Syntax checks for Playbooks:

#ansible-playbook --**syntax-check** play1.yml

[root@ansible]#ansible-playbook --syntax-check play1.yml
ERROR! Syntax Error while loading YAML.

The error appears to have been in '/etc/ansible/play1.yml': line 3, column 8, but may be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

0
- hosts: all
    ^ here

Playbooks Dry Run

Dry Runcheck for playbooks:

```
# ansible-playbook  <playbook-name>  --check

# [root@Ansible]#ansible-playbook play1.yml --check
SSH password:
PLAY [all] ********************************************************************
TASK [setup] ******************************************************************
ok: [192.168.74.51]
TASK [File Creation] **********************************************************
changed: [192.168.74.51]
TASK [Directory Creation] ******************************************************
changed: [192.168.74.51]
PLAY RECAP ********************************************************************
192.168.74.51              : ok=3    changed=2    unreachable=0    failed=0
```

Playbooks Real Time Run

Real time run for playbooks:

```
# ansible-playbook play1.yml
SSH password:
PLAY [all] *********************************************************************
TASK [setup] *******************************************************************
ok: [192.168.74.51]
TASK [File Creation] ***********************************************************
changed: [192.168.74.51]
TASK [Directory Creation] ******************************************************
changed: [192.168.74.51]
PLAY RECAP *********************************************************************
192.168.74.51              : ok=3    changed=2    unreachable=0    failed=0
```

Lab

- Create your first Playbook to create a user name "test" with user ID 8888 and a directory name "/var/tmp/test".

- Create a Playbook for User and Group Creation with user name "usertest", shell bash, userid 6666 and pass the comments as "my first user". Group details will be name "grouptest" and group id 7777.

- Create a Playbook for files and directories:

- create a directory with root ownership; inside this directory, create one file with “test” ownership (the user we have created in 1st example). Then finally copy some log files (say /var/log/messages) to the newly created file.

Step Run

Moving through task by task execution:

Special Case run Ansible playbooks with –step

Example:

```
[root@user20-master plays]# ansible-playbook ntp.yaml --step
PLAY [all] ********************************************************************************************************************
Perform task: TASK: Gathering Facts (N)o/(y)es/(c)ontinue: n
Perform task: TASK: Second task (N)o/(y)es/(c)ontinue:
**********************************************************************
Perform task: TASK: Third task to start NTP Services (N)o/(y)es/(c)ontinue: y
TASK [Third task to start NTP Services] ***************************************************************************************
ok: [user20-client]
PLAY RECAP
********************************************************************************************************************
user20-client          : ok=1    changed=0    unreachable=0    failed=0
[root@user20-master plays]#
```

Handlers

- In simple layman language Handlers are **“Running Operations on Changes”.**
- Handlers are list of tasks with unique names.
- “notify” is the keyword to trigger operations mentioned in Handlers.
- Regardless of how many tasks notify a handler, it will run only once, after all of the tasks complete in a particular play.
- If nothing changed in the tasks, Handlers will not run.
- Multiple “notify” with unique names are permitted.
- Handlers are always mentioned at last.

Handlers

Let's understand the importance of Handlers with below use cases:

- Playbook without Handlers
- Playbook with Handlers (with improper usage of handler)
- Playbook with Proper use of Handler

Lab - NTP

- Let’s do a practical for NTP module including:

- Package
- Files
- Service

Lab - NTP

```
[root@Ansible]#cat ntp.yml
---
- hosts: all
  tasks:
    - name: NTP OS Package Installation
      package: name=ntp state=present
    - name: NTP File Configurations
      file: path=/etc/ntp.conf state=file
    - name: To start NTP services
      service: name=ntpd state=started enabled=yes
```

Lab - NTP

```
[root@sandeep]#ansible-playbook ntp.yml --ask-pass
SSH password:

PLAY [all] ********************************************************************

TASK [setup] ******************************************************************
ok: [192.168.74.51]

TASK [NTP OS Package Installation] ********************************************
changed: [192.168.74.51]

TASK [NTP File Configurations] ************************************************
ok: [192.168.74.51]

TASK [To start NTP services] **************************************************
changed: [192.168.74.51]

PLAY RECAP ********************************************************************
192.168.74.51              : ok=4    changed=2    unreachable=0    failed=0
```

NTP without proper config

```
[root]#cat ntp.yml
---
- hosts: all
  tasks:
    - name: NTP OS Package Installation
      package: name=ntp state=present
    - name: NTP File Configurations
      file: path=/etc/ntp.conf state=file
      notify:
      - restart ntp
    - name: To start NTP services
      service: name=ntpd state=started enabled=yes
  handlers:
    - name: restart ntp
      service: name=ntpd state=restarted
```

Now run the Ansible Playbook and See nothing will change as no configuration is getting changed.

NTP with proper config

```
[root]#cat ntp.yml
---
- hosts: all
  tasks:
    - name: NTP OS Package Installation
      package: name=ntp state=present
    - name: NTP File Configurations
      copy: src=/etc/ntp.conf dest=/etc/ntp.conf
      notify:
      - restart ntp
    - name: To start NTP services
      service: name=ntpd state=started enabled=yes
  handlers:
    - name: restart ntp
      service: name=ntpd state=restarted
```

Ansible Variable overview

- Like any other Automation language Ansible also supports variables.
- In layman language variable is just an assigned value (string or characters).
- Variables can be used with any playbook, tasks, roles, templates and even with inventory files.
- There is a well-defined variable naming scheme in Ansible, it should always start with alphabet.
- Variables are first defined and then declared to be used in the Ansible.
- In simplest way variables are defined inside playbooks under “vars” section in the play.

Ansible Variable overview

- Variable Definition
- # Defining variable section in playbook using vars parameter
- vars:
- user1: sandeep

- Variable Declaration
- tasks:
- - name: user creation
- user: name={{ user1 }} state=present

Ansible Variable overview

You can define empty variables in playbook and later on provide variable values during runtime:

```
[root@user20-master plays]# cat variable-test1.yaml
---
- name: Starting a test play for Variable demo
  hosts: all
  vars:
    user1: sandeep
    user2: ""
    uid1: 9876
    uid2: 8765
  tasks:
    - name: Creating a user1
      user: name={{user1}} uid={{uid1}} state=present
    - name: Creating user {{user2}}
      user: name={{user2}} uid={{uid2}} state=present
[root@user20-master plays]#
[root@user20-master plays]# ansible-playbook variable-test1.yaml -e user2=“sandeep"
```

Ansible Facts

- Facts are the special variables automatically discovered by Ansible.
- Dedicated “setup” module.
- Facts output is in JSON format.
- Facts provide information about hostname, kernel, network, OS and much more.
- The information from facts can be filtered using -a filter=<value> option to get specific information.

Ansible Facts

- [root@controller ~]# ansible [host] -m setup
- [root@controller ~]# ansible [host] -m setup | grep -i "ansible_"
- [root@controller ~]# ansible [host] -m setup -a filter="ansible_all_ipv4_addresses"
- [root@controller ~]# ansible [host] -m setup -a filter="ansible_date_time"

Ansible Facts

- Facts values are used as variables to create useful generic and dynamic playbooks .
- Facts outputs are JSON in format.
- Declared format in playbook is {{ <ansible_facts> }}
- Ansible automatically replace the Facts variable with values while Ansible run.

Ansible Facts

- First step is to find facts to be used in Playbook.
- [root@controller ~]# ansible centos-managed -m setup -a filter="ansible_hostname“
- [root@controller ~]# ansible "*" -m setup -a filter="ansible_eth0“

Note: "ansible_eth0"will be having detailed output in JSON format which you can even filter out for specific value. In our example we will filter out “interface address” as {{ ansible_eth0.ipv4.address }} which means check ansible_eth0 then go to ipv4 section and then return the address attribute value.

Ansible Facts

```
[root@]# cat factstest.yml
---
- name: This is my First Debug Play
  hosts: all
  tasks:
    - name: Testing Ansible Facts {{ ansible_hostname }}
# My task with outputs fetched from Facts
      debug: msg="Host {{ ansible_hostname }} is having IP address {{ ansible_eth0.ipv4.address }}"
...
```

Lab

- Install an httpd package in playbook
- Configure hostname and IP address in webpage (using facts) and owner (through variable)
- Handler to be used for restartng httpd service

Lab

- Create a playbook for to install httpd (apache webserver

```
---
- name: Apache webserver deployment
  hosts: client

  tasks:
  - name: Installing Apache packages
    yum: name=httpd state=present

  - name: Changing Listening port to 81
    lineinfile:
      path: /etc/httpd/conf/httpd.conf
      regexp: "Listen 80"
      line: Listen 81

  - name: Configuring Apache configuration files
    copy: dest=/var/www/html/index.html content="<h1>This is a demo for Apache Webserver!</h1>“

 - name: Starting Apache webservice
    service: name=httpd enabled=yes state=started
```

Ansible Verbose Run

Run Ansible in Verbose mode to show detailed execution logs

```
[root@user20-master plays]# ansible-playbook debug-register.yml -vv
ansible-playbook 2.4.2.0
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules',
u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /bin/ansible-playbook
  python version = 2.7.5 (default, Oct 30 2018, 23:45:53) [GCC 4.8.5 20150623 (Red Hat 4.8.5-36)]
Using /etc/ansible/ansible.cfg as config file

[root@user20-master plays]# ansible-playbook debug-register.yml  | wc -l
21
[root@user20-master plays]# ansible-playbook debug-register.yml -vvvvv | wc -l
233
```

Ansible Debug

Debug Module is used for debugging the outputs.

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

Parameter	Choices/Defaults	Comments
msg	**Default: Hello world!**	The customized message that is printed. If omitted, prints a generic message.
var		A variable name to debug. Mutually exclusive with the 'msg' option.
verbosity	**Default: 0**	A number that controls when the debug is run, if you set to 3 it will only run debug when -vvv or above

Ansible Debug

```
[root]# cat debug.yaml
---
- name: Debug Playbook
  hosts: all
  tasks:
    - debug:
        msg: First task to Print debug message

    - debug:
        msg: "System {{ inventory_hostname }} has uuid {{ ansible_product_uuid }}"
[root]#
```

Ansible Register

Ansible comes with a special variable name “Register” to capture command output run by Ansible run and save them into variable value.

It is mostly used with Debug Module.

```
[root]# cat debug-register.yaml
---
- name: Debug Playbook
  hosts: all
  tasks:
    - name: Talk package installation
      yum: name=telnet state=present
      register: output

    - name: Running debugger to show the output
      debug:
        var: output

[root]#
```

Register - Printing specific value

Specific value from Ansible Register output can also be printed:

```
---
- name: This is my First Debug Play
  hosts: all
  tasks:
    - name: install ntp
      yum: name=ntp state=installed
      register: ntp_out

    - name: printing complete output
      debug: var=ntp_out

    - name: printing specific reults for individual parameter
      debug: var=ntp_out.changed   // such parameter can be seen in output from previous task output
```

Ansible Ignore Errors

By default if a task fails, Ansible stops running further playbook tasks and exits the program.

When you want to ignore failure for a task, which don't have any impact on pending tasks, you can ignore errors for that task.

```
[root]# cat ignore_error.yaml
---
- name: Ignore Error Playbook
  hosts: all
  tasks:
    - name: This will not be counted as failure
      command: /bin/date12
      ignore_errors: yes

    - name: Second task
      Service: name=ntpd state=restarted

[root]#
```

Working with Conditions

Conditional tasks can be performed using “When” Statement.

Ansible allows special statements with “AND” and “OR”.

```
---
- name: Conditional checks
  hosts: all
  tasks:
    - name: Installing web packages on all centos version 7 servers
      yum: name=httpd state=installed
      when: ansible_distribution == “CentOS” and ansible_distribution_major_version == “7”

 - name: "shut down CentOS 6 and Debian 7 systems"
    command: /sbin/shutdown -t now
    when: (ansible_facts['distribution'] == "CentOS" and ansible_facts['distribution_major_version'] == "6") or
        (ansible_facts['distribution'] == "Debian" and ansible_facts['distribution_major_version'] == "7")
```

Working with Conditions

Working based on previous task output

```
- name: Register a variable
  package: name=ntp state=installed
  register: ntp_out
  ignore_errors: true
- name: debug
  debug:
    var: ntp_out
- name: Use the variable in conditional statement
  shell: echo "motd contains the word ansible"
  when: ntp_out.rc == 0

- name: Register a variable
  shell: cat /etc/motd
  register: motd_contents

- name: Use the variable in conditional statement
  shell: echo "motd contains the word hi"
  when: motd_contents.stdout.find('hi') != -1
```

Working with loops

Often, you'll want to do many things in one task, such as create a lot of users, install a lot of packages, or repeat a polling step until a certain result is reached.

Handling multiple similar tasks simultaneously can be done in Ansible to keep your code small and simple.

php gcc talk vim httpd

use **with_items or loop**

```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  with_items:
     - testuser1
     - testuser2
```

Working with loops

e.g

```
- file:
    src: '/tmp/{{ item.src }}'
    dest: '{{ item.dest }}'
    state: link
  with_items:
    - { src: 'x', dest: 'y' }
    - { src: 'z', dest: 'k' }
```

Use Loop instead of with_items

With_item is being replaced with loop now.

```
- name: add several users
  user:
    name: "{{ item }}"
    state: present
    groups: "wheel"
  loop:
     - testuser1
     - testuser2
```

Working with loop and condition

Re-iteration with condition:

```
- name: Run with items greater than 5
    command: echo {{ item }}
    loop: [ 0, 2, 4, 6, 8, 10 ]
    when: item > 5
```

Reading variables/item from files:

```
 - name: task to create multiple users
    user: name={{ item }} state=absent
    with_lines: cat /root/gagan/users.txt
```

Lab

Install below packages on CentOS version 7 machine only:

php

gcc

talk

vim

httpd

Used conditions and Loops

Reboot Machines during a play

```
- name: Run with items greater than 5
  command: echo {{ item }}
  loop: [ 0, 2, 4, 6, 8, 10 ]
  when: item > 5

- name: Reboot a slow machine that might have lots of updates to apply
  reboot:
    reboot_timeout: 300

- name: second task for package
  package:
    name: telnet
    state: absent
```

Retries

We can retry failed task on a specific task:

```
- name: second task for package
  package:
    name: telnet
    state: absent
  retries: 2
```

Limiting playbook to limited servers can be done via:

ansible-playbook first.yml --**limit** client

Ansible Blocks

Blocks create logical groups of tasks. Blocks also offer ways to handle task errors, similar to exception handling in many programming languages.

```
tasks:
  - name: Install, configure, and start Apache
    block:
      - name: Install httpd and memcached
        yum:
          name:
          - httpd
          - memcached
          state: present

      - name: Apply the foo config template
          template:
          src: templates/src.j2
          dest: /etc/foo.conf

      - name: Start service bar and enable it
        service:
          name: bar
          state: started
          enabled: True
    when: ansible_facts['distribution'] == 'CentOS'
    become: true
    become_user: root
    ignore_errors: yes
```

Ansible Blocks

Blocks create logical groups of tasks. Blocks also offer ways to handle task errors, similar to exception handling in many programming languages.

```
[root@gagan-ansible ~]# cat rescue.yml
---
- name: rescure example
  hosts: gagan-client

  tasks:
    - name: Handle the error
      block:
        - name: Print a message
          debug:
            msg: 'I execute normally'

#       - name: Force a failure
#         command: /bin/false

        - name: Never print this
          debug:
            msg: 'I never execute, due to the above task failing, :-('
      rescue:
        - name: Print when errors
          debug:
            msg: 'I caught an error, can do stuff here to fix it, :-)'

        - name: Print when errors - task 2
          debug:
            msg: 'second task to work with :-)'

      always:
        - name: Always do this
          debug:
            msg: "This always executes"
[root@gagan-ansible ~]#
```

Best Practices

Logs

By Default, logging is disabled. Enable same for audit purpose and for storing the output for a later reuse.

Uncomment below logging configuration in ansible.cfg file:

log_path = /var/log/ansible.log

To disable tasks logs at master/client level, below parameters can be touched:

prevents logging of task data, off by default

#no_log = True

prevents logging of tasks, but only on the targets, data is still logged on the master/controller

#no_target_syslog = False

#By default, logs goes as per syslog configuration on client machine.

Working with tags

Tags helps us to **run/avoid** specific **tasks/blocks/roles** from a playbook.

```
 - name: first task for creating a user
     user:
       name: samdeep
       state: present
       tags:
         - prod

    - name: Run with items greater than 5
      command: echo {{ item }}
      loop: [ 0, 2, 4, 6, 8, 10 ]
      when: item > 5
      tags:
        - dev
```

```
# ansible-playbook first.yml --tags dev
# ansible-playbook first.yml --tags prod,db
# ansible-playbook first.yml --skip-tags prod
```

You can also use **--start-at-task** for running playbook from a specific task.

Disable Gather_facts if not required

You can disable the facts in playbook to avoid network congestion due to Facts collection feature of Ansible before every node run:

```
---
- name: This is my First Debug Play
  hosts: all
  gather_facts: no

  tasks:
    - name: Testing Ansible Facts {{ ansible_hostname }}
      debug: msg="Host {{ ansible_hostname }} is having IP address {{ ansible_eth0.ipv4.address }}"
...
```

Note: This playbook will fail as we are using environment variables without fetching the facts