Ansible

Introduction to Ansible

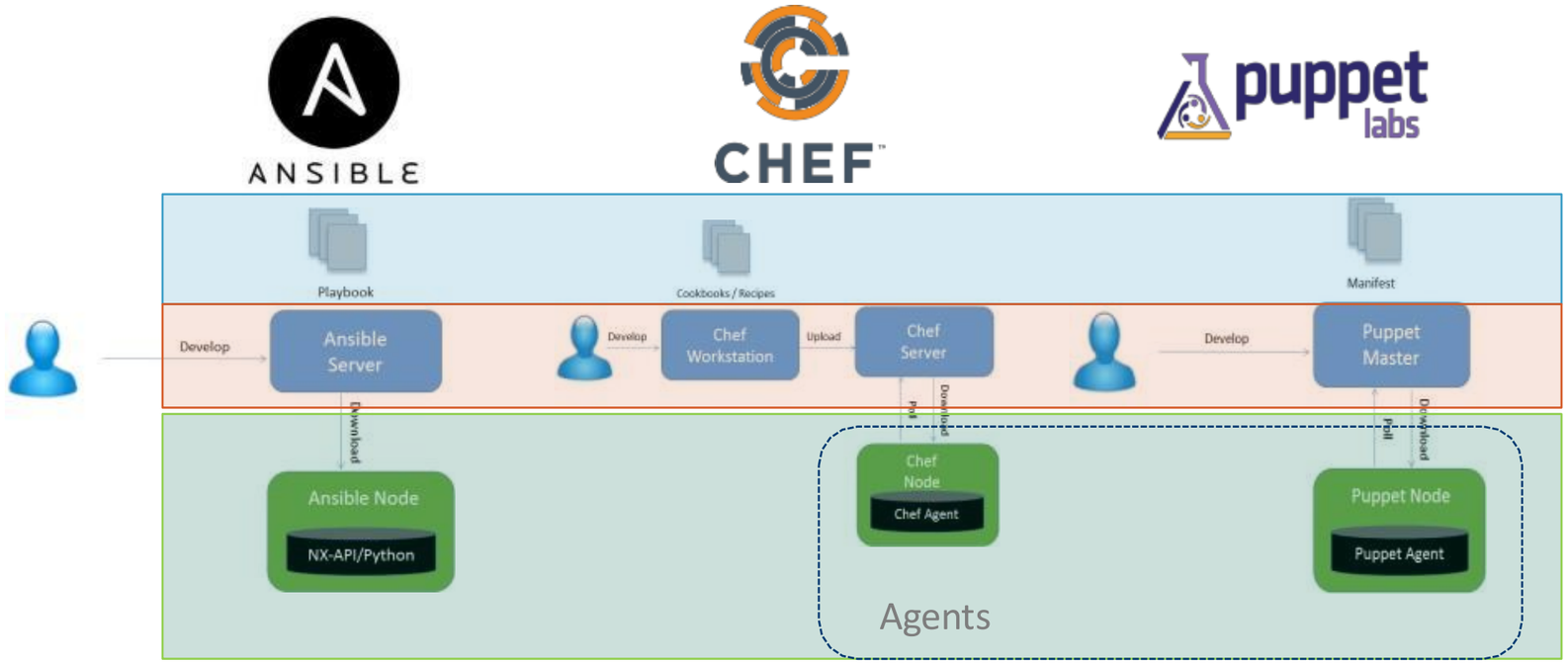
Module 1

Why Configuration Management Matters

Agent-based Agentless

- Speed
- Automation
- Repeatability
- Mitigate Configuration Errors
- Normalization
- Programable Infrastructure
- Human readable code
- Version control
- Idempotency
- Integration with other business processes

1. Set of “Tasks”
2. Master Server
3. Nodes



	Chef	Puppet	Ansible
Template	Cookbook/Recipe	Manifest	Playbook
Language	Extension of Ruby	Custom JSON-like language with Ruby option	YAML
License	Apache	Apache (earlier versions are GPL)	GPL
Agent	Required	Required	Not required
Idempotent	Yes	Yes	Yes

	Puppet	Chef	Salt	Ansible
Initial release	2005	2009	2011	**2012**
Configuration Language	DSL	Ruby/DSL	YAML	**YAML**
Template Language	ERB	ERB	Jinja2	**Jinja2**
Agentless				✓
Simple ad-hoc task execution			✓	✓
GitHub ★s*	2,239	2,729	3,531	**6,202**

```
...
cisco_package "#{name}/n9000-
dk9.6.1.2.I3.1.CSCur02700.bin
" do

  source "bootflash:/n9000-
dk9.6.1.2.I3.1.CSCur02700.bin
"
  action :make_active
end
...
```

```
...
class
cisco_onep::interface_config
{

  cisco_vlan { "$::hostname
20":
    ensure => present,
    vlan_name =>
"puppet_VLAN_20",
    state => active
  }

  cisco_interface {
"$::hostname Ethernet1/16":
    switchport => 'trunk',
    description => 'puppet',
    access_vlan => 20
  }
...
```

Ansible Playbook


```
---

- name: sample playbook
  hosts: leaf

  tasks:

  # Example of nxos_switchport module that configures e1/2 on port vlan 20
- nxos_switchport: interface=Ethernet1/2 mode=access access_vlan=20 host={{
inventory_hostname}}

  # Example of nxos_command module that passes CLI commands
- nxos_command:
     host: "{{ inventory_hostname}}"
     type: config
     command: ['hostname N9K-Standalone-Pod-1']
```

Why Ansible?

What is Ansible?

- Radically simple IT automation engine that automates
  - Cloud provisioning
  - Configuration management
  - Application deployment
  - Intra-service orchestration

Why Ansible?

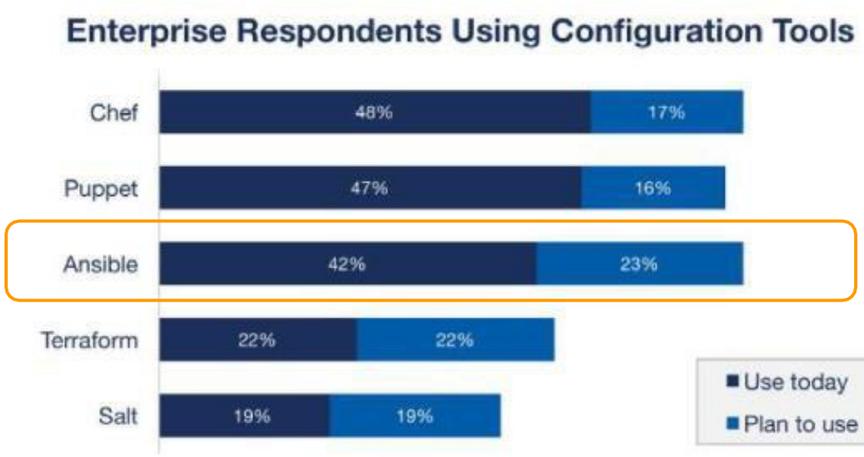
- Simple
  - Easy to write, read, maintain and evolve- without writing scripts or custom code
- Fast to learn and setup
  - It uses a very simple language (YAML, in the form of Ansible Playbooks) that allow you to describe your automation jobs in a way that approaches plain English.

Why Ansible?

- Efficient
  - Doesn't require a custom agent or software to install
  - Ansible works by connecting to your nodes and pushing out small programs, called "Ansible modules" to them.
- Secure
  - No agent
  - Runs on OpenSSH

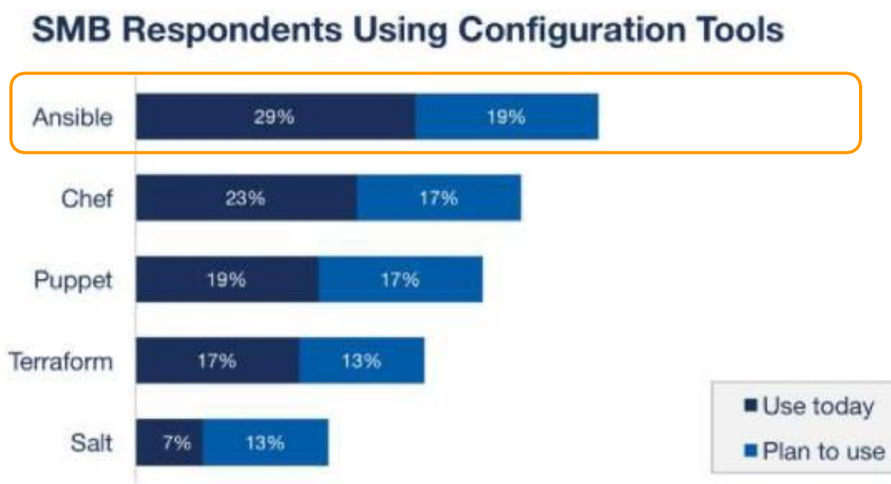
RightScale State of the Cloud Report

ENTERPRISE ADOPTION



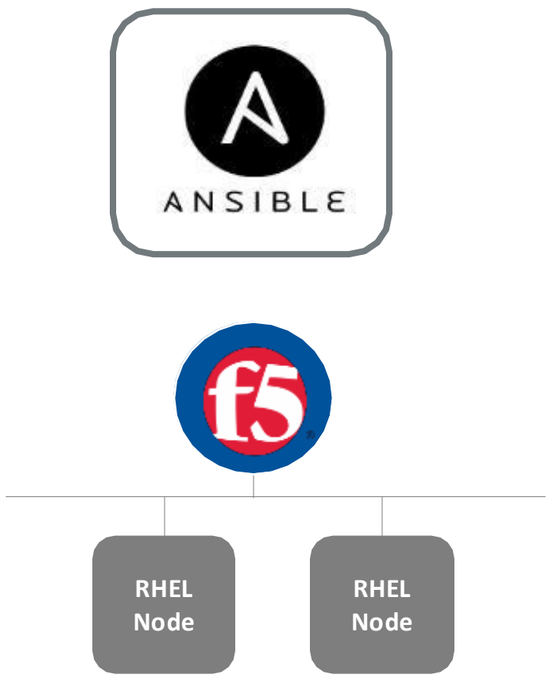
Source: RightScale 2018 State of the Cloud Report

SMB ADOPTION



Source: RightScale 2018 State of the Cloud Report

Use Case: Single Device Provisioning



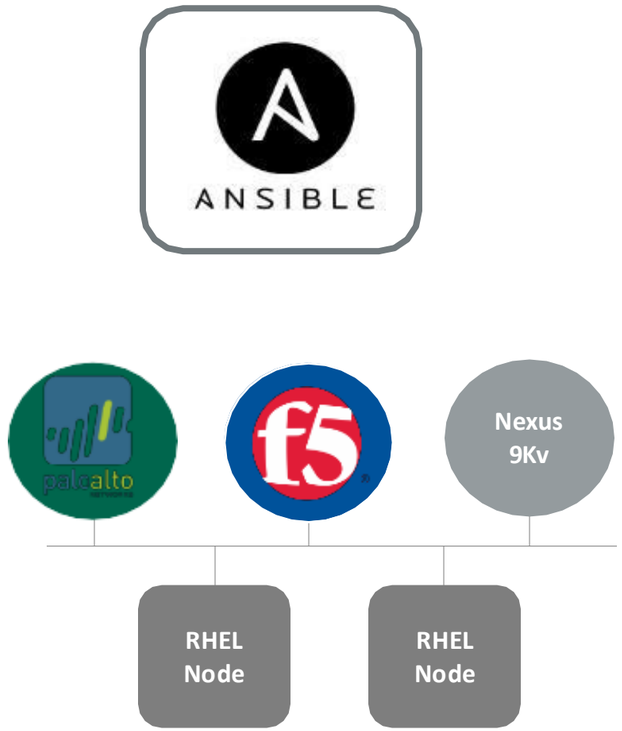
Load Balancer Provisioning Overview

- Provisioning tasks on the F5 load balancer is time intensive
- Many of the mundane tasks can be automated.

Use Case Example Tasks

- Configuring VLANs
- Configuring Self-IPs
- Create nodes with health monitors
- Create pool
- Create SNAT pool
- Assign members to pool and SNAT pool
- Create a HTTPS virtual server

Use Case: Multi-vendor Device Provisioning



Load Balancer plus Network & Firewall Provisioning Overview

- Manual provisioning of a single device helps save time
  - Biggest business benefits are when configurations can be applied consistently across multiple devices across the enterprise

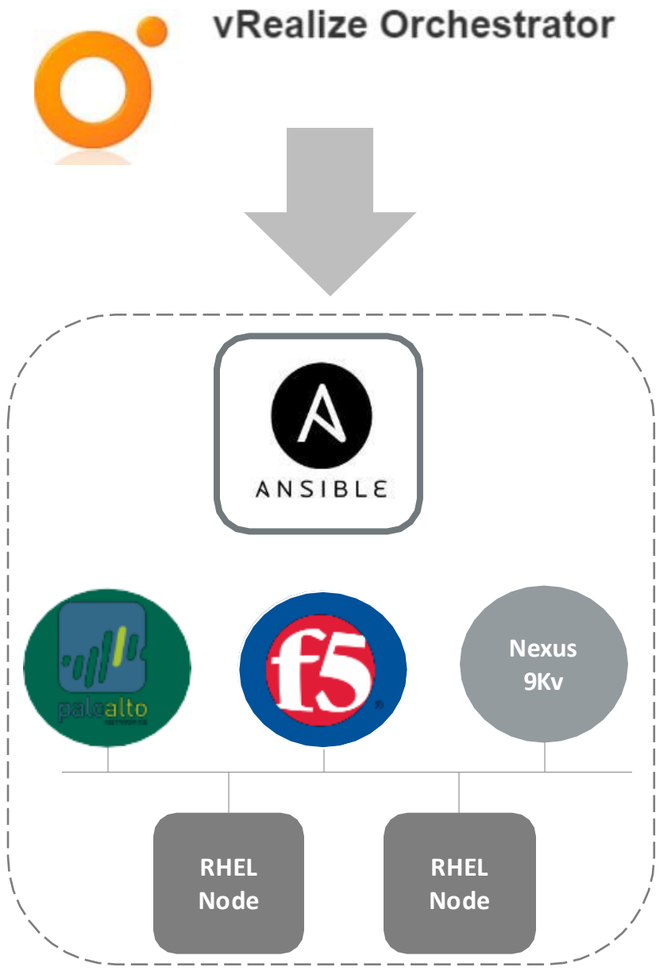
1

1

Use Case Example Tasks

- Leverage existing load balancer playbook
- Combine playbooks for provisioning network security such as:
  - Access-lists on Cisco N9kv
  - Firewall rules (e.g. zones and simple policies applied to Palo Alto)

Use Case: Orchestration



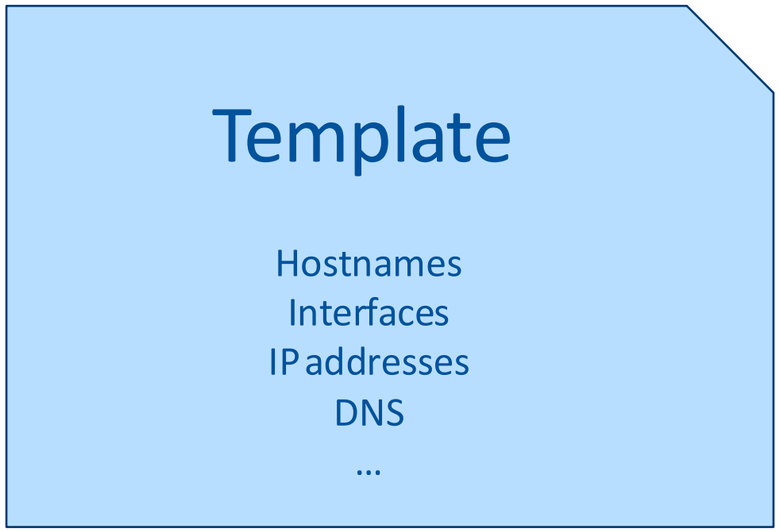
Integrations with 3rd Party Tools

- In addition to simplifying the automation of multi-vendor infrastructure environments from a single controller,
  - Ansible can act as an extension to existing IT Service Management platforms to accelerate business service requests.

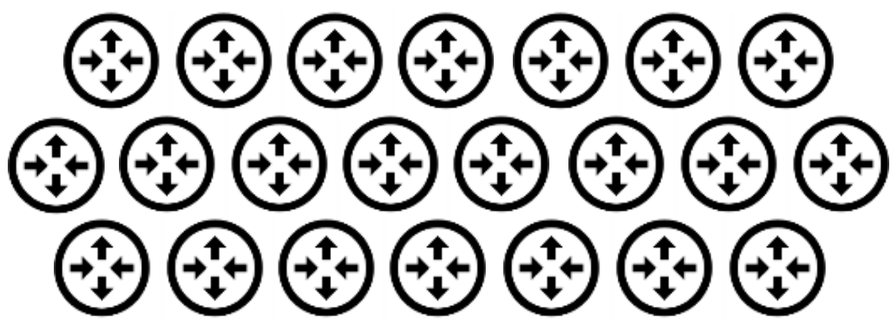
Use Case Example Task

- vRO remotely executes a set of Ansible commands which automates the provisioning of infrastructure services (e.g. network, firewall, load balancer)

Use Case: Templatized Configuration



Variable Substitution
and
Looping



More Use Cases Beyond Device Provisioning

- Application Deployment
- Continuous Delivery
- Security and Compliance

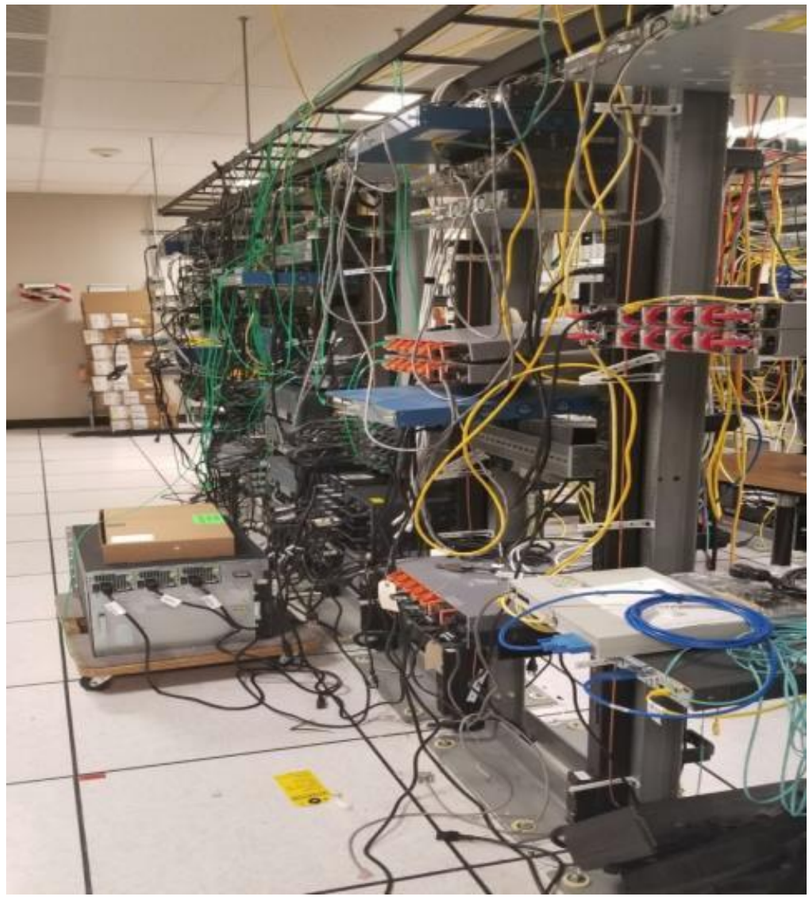
Resources

Ansible Best Practices	https://docs.ansible.com/ansible/2.5/user_guide/playbooks_best_practices.html
Ansible Docs	https://docs.ansible.com/
Ansible Network Modules	https://docs.ansible.com/ansible/2.6/modules/list_of_network_modules.html
Ansible Variables	https://docs.ansible.com/ansible/2.6/user_guide/playbooks_variables.html
Ansible Galaxy	https://galaxy.ansible.com/home
Jinja	http://jinja.pocoo.org/

Thank You

Real World Use Cases

Module 2



Look Familiar?

Common Theme Across IT Industry

Long lead times due to tasks that are:

- Manual and repetitive
- Siloed
- Not human readable and prone to human error
- Must be performed against large groups of devices
- Must be integrated with other business processes (eg: vRO, ServiceNow, and so on)

Need for Change Validation

- How can I see changes that will be made prior to committing?

Real World Use Cases

Single Device Provisioning

Multi-vendor Device Provisioning

Orchestration

Templated Configurations

VXLAN Configuration

Problem

VXLAN takes time to learn and setup for network engineers exposed to the capability the first time

Use Case

Bring up VXLAN with BGP EVPN with little experience

Advantages

Save time

Get VXLAN BGP EVPN up and running quickly

Learn VXLAN by playing with a tested setup

Custom Command

PRGM

Problem

NOC engineers often issue multiple commands just to get simple details from the network. This process is manual and repetitive.

Use Case

Display specific details from neighbour using a single programmatic Python instruction versus a series of CLI commands

Advantages

Single command

Speed

Consistency

Information you only care about

LLDP Based Interface Descriptions

PRGM

Problem

Network engineers would like to maintain good interface descriptions but it takes time, especially, when there are lot of interfaces to manage

Use Case

Use LLDP information to programmability add interface descriptions

Advantages

Consistent

Removes human error (eg: fat fingering)

Saves time

OSPF and PIM Interfaces

PRGM

Problem

OSPF and PIM interface configuration can be burdensome and error prone, especially, when there are lots of interfaces to configure

Use Case

Provide menu driven options for junior engineers to configure OSPF and PIM interface attributes

Advantages

CLI expertise not required

Human error removed

Consistent

Time saved

Troubleshooting CPU Utilization

TSHOOT

Problem

Checking the CPU health of a switch is a repetitive task and often only a portion of the information displayed is relevant

Use Case

Check CPU health of switch at regular intervals and log output to file for TAC

Advantages

Extract relevant information

Automate repetitive tasks

Troubleshooting Memory Utilization

TSHOOT

Problem

Manually checking memory usage on a switch at regular intervals is repetitive and often too verbose

Use Case

Check memory utilization of switch at regular intervals and generate log for TAC

Advantages

Extract relevant information

Automate repetitive tasks

Troubleshooting Traffic Loss Issues

TSHOOT

Problem

When there are problems with traffic flow in the network, network engineers have to use multiple commands at different intervals for checking traffic flow through an interface.

Use Case

Check RX packet count from an interface at regular intervals and generate log for TAC.

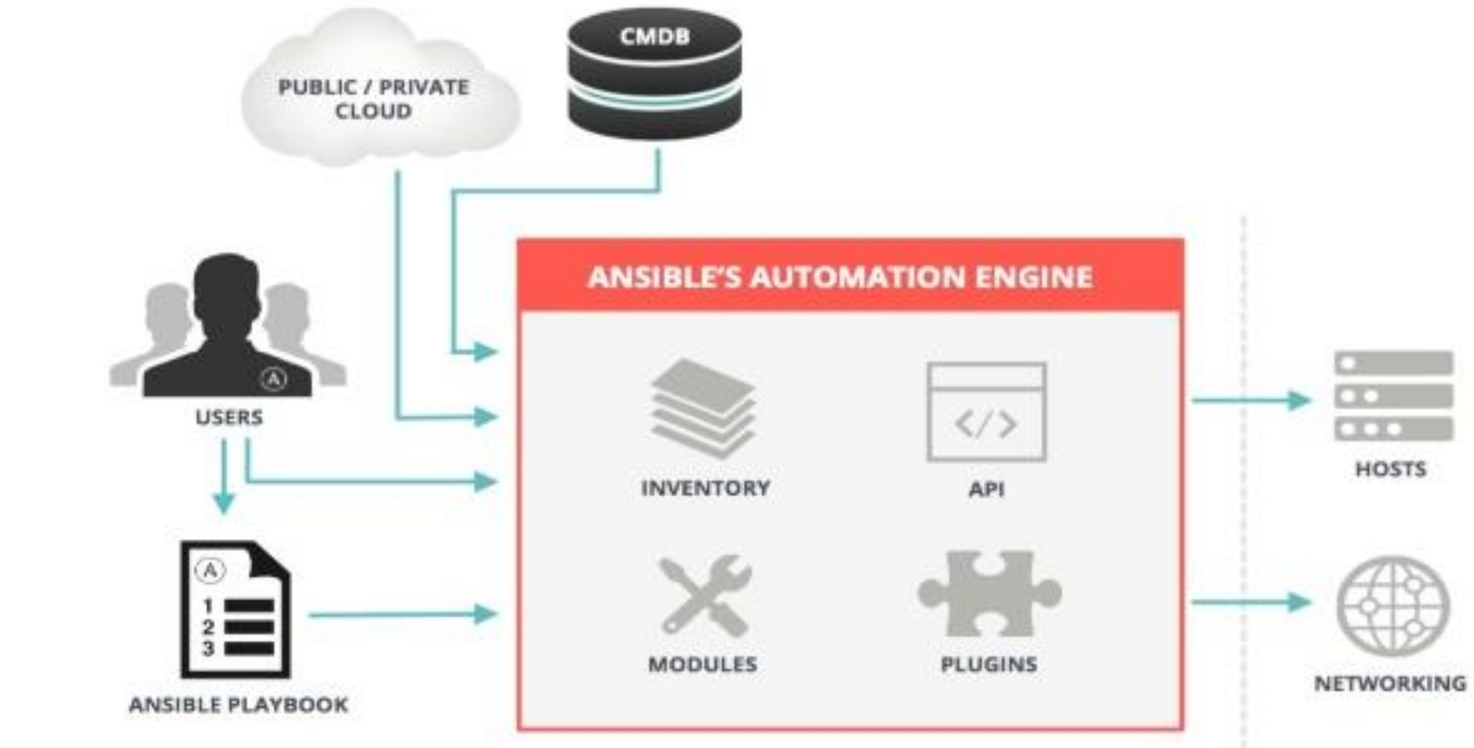
Advantage

Execute commands programmatically to check for traffic drops/errors.

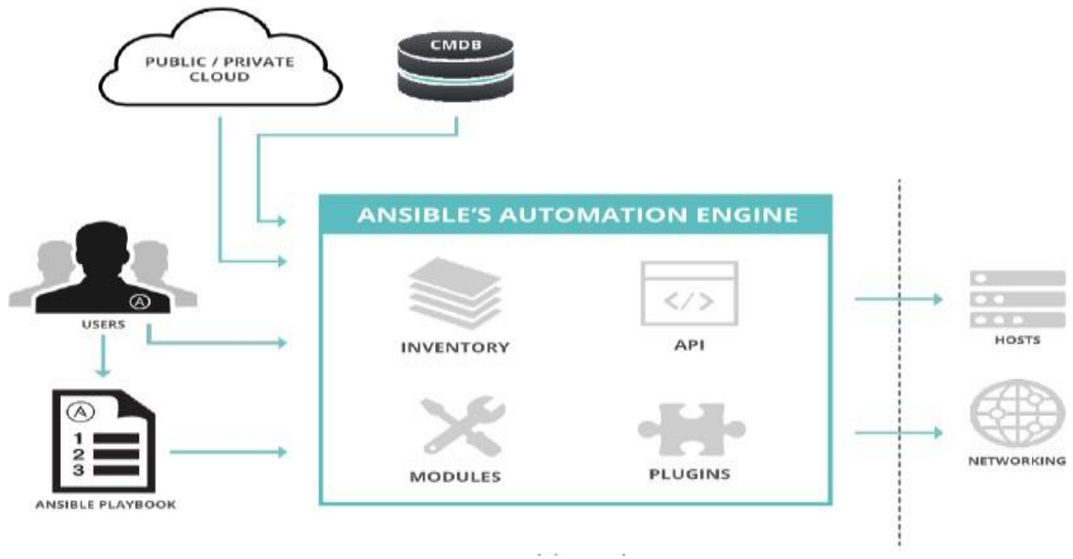
Ansible Architecture

How Ansible Works?

Ansible works by connecting to your nodes and pushing out small programs, called **“Ansible Modules”** to them. Ansible then executes these modules **(over SSH by default)** and removes then when finished.



Ansible Architecture and Concepts



Ansible Nodes

Control Node
Managed Hosts
Managed Networking

Inventory

Static Inventory
Dynamic Inventory

Ansible Configuration file

Ad Hoc Commands

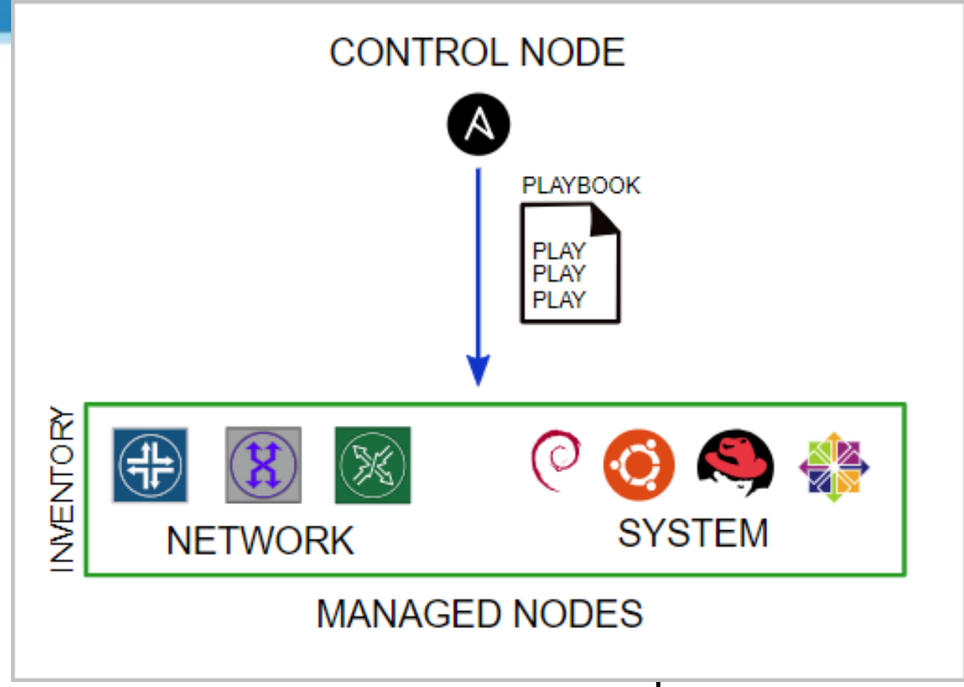
Playbook

YAML
Hosts
Plays
Tasks

Modules

Ansible Nodes

- **Control Node**
  - **Ansible** and **python** installed machine
  - The host on which you use Ansible to **execute tasks** on the managed nodes
- **Managed Hosts**
  - Ansible **not** installed , but python installed machines
  - A host that is **configured** by the control node using standard network protocol
- **Managed Networking**
  - Ansible can manage network devices via ssh or custom modules supplied by vendors
  - Some modules already built in consist of Nexus, Juniper, Arista, and more.



control machine

Python ≥ 2.6/2.7

Ansible:

- *pip install ansible*
- *yum install ansible*
- *apt-get install ansible*
- *brew install ansible*

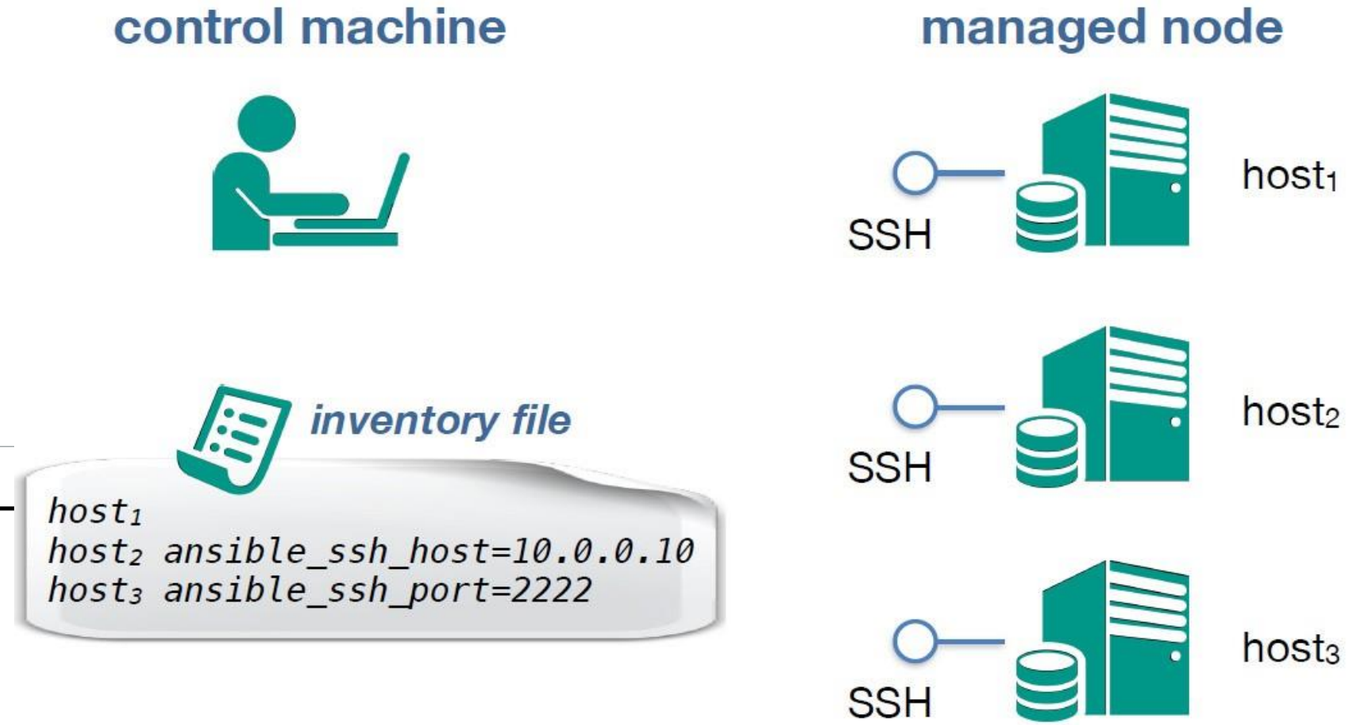
managed node

Python ≥ 2.5

Ansible Inventory

Inventory is a collection of hosts/network devices (nodes) with associated data and groupings that Ansible can connect and manage.

+ Hosts (nodes)
+ Groups
+ Inventory-specific data (variables)
+ Static Inventory
+ Dynamic Inventory



Inventory

- Ansible works against multiple systems in your infrastructure at the same time. It does this by selecting portions of systemslisted in Ansible's inventory file, which defaults to being saved in the location /etc/ansible/hosts.

[webservers]	[appservers]	[dbservers]
192.168.35.140	192.168.100.1	172.35.0.5
192.168.35.141	192.168.100.2	
192.168.35.142	192.168.100.3	
192.168.35.143		

Host

A host is simply a remote machine that Ansible manages. They can have individual variables assigned to them, and can also be organized in groups.

```
[webservers]
192.168.35.140
192.168.35.141
192.168.35.142
192.168.35.143

[appservers]
192.168.100.1
192.168.100.2
192.168.100.3

[dbservers]
172.35.0.5
```

Group

- A group consists of several hosts assigned to a pool that can be conveniently targeted together, and also given variables that they share in common.

```
[webservers]
192.168.35.140
192.168.35.141
192.168.35.142
192.168.35.143

[appservers]
192.168.100.1
192.168.100.2
192.168.100.3

[dbservers]
172.35.0.5
```

Static Inventory Examples

[master]

podx-master.origin.com

[nodes]

podx-node1.origin.com

podx-node2.origin.com

172.1.120.xx

[everyone:children]

master

nodes

```
test.lab.local

[sydney]
host1.lab.sydney
host2.lab.sydney

[melbourne]
host1.lab.melbourne
host2.lab.melbourne

[australia:children]
sydney
melbourne
```

```
[production:children]
webservers
dbservers
proxies

[webservers]
foo.example.com http_port=80
bar.example.com http_port=8080

[dbservers]
db[01:03].example.com

[dbservers:vars]
pgsql_bind_nic=eth1

[proxies]
192.168.1.1
```

Ansible Configuration File

Ansible installation can be customized by modifying settings in the Ansible configuration file located in 3 possible locations:

1	Base Config	**/etc/ansible/ansible.cfg**
2	User's Home Directory	**~/.ansible.cfg**
3	Directory Ansible Command is Executed	**./ansible.cfg**

Note: If an ansible.cfg file exists in the directory in which the ansible command is executed, it is used instead of the global file or the user's personal file.

Ansible Configuration file Example

cat /etc/ansible/ansible.cfg

[defaults]
inventory = ./inventory
remote_user = <someuser>
ask_pass = false

[privilege_escalation]
become = true
become_method = sudo
become_user = root
become_ask_pass = false

Become and Network Devices

- As of version 2.6 now supports become for entering enable mode or priv EXEC mode on ansible maintained platforms that support enable.
  - Examples are eos, ios, and nxos. “become” now replaces the authorize and auth_pass that you see in the provider options.
- Use you can use these by specifying become: yes and become_method: enable to instruct Ansible to enter enable mode.

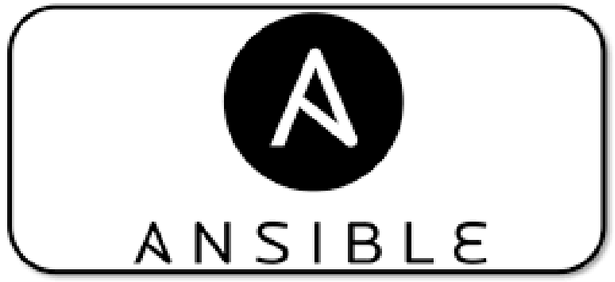
Become Example

```
-hosts: eos-switches
    become: yes
    become_method: enable
    tasks:
      - name: Gather facts (eos)
        eos_facts: gather_subset:
                   - "!hardware"
```

Ad hoc command

An ad-hoc command is a single Ansible task to perform quickly, but don't want to save for later.

```
# check all my inventory hosts are ready to be managed by
#Ansible
$ ansible all -m ping

# collect and display the discovered facts for the
localhost
$ ansible localhost -m setup

# run the uptime command on all hosts in the web group
$ ansible web -m command -a "uptime"
```



AD-HOC
COMMANDS

inventory file

```
ansible                                  \
    --inventory-file=hosts-vagrant       \
    --user=vagrant --ask-pass            \
    all                                  \
    -a hostname
```

user account

apply to “all” hosts in the inventory file

ad-hoc command

Playbook, Plays, Tasks, and YAML

- Playbooks are a file containing one or more plays.
- Plays are ordered sets of tasks to execute against host selections from your inventory.
- Tasks are actions performed within a play.
- YAML is the language Ansible uses to build playbooks.

Playbook

- Playbooks are the language by which Ansible orchestrates, configures, administers, or deploys systems. Playbooks contain Plays.

Install application server and database server

Install & Start Apache Tomcat

Install Java

Install Tomcat

Install & Start MySQL & Import Data

Install MySQL

Import Data

Play

A play is a mapping between a set of hosts selected by a host specifier and the tasks which run on those hosts to define the role that those systems will perform.

Install application server and database server

Install & Start Apache Tomcat

Install Java

Install Tomcat

Install & Start MySQL & Import Data

Install MySQL

Import Data

Task

Tasks combine an action with a name and optionally some other keywords (like looping directives). Tasks call modules .

Install application server and database server

Install & Start Apache Tomcat

Install Java

Install Tomcat

Install & Start MySQL & Import Data

Install MySQL

Import Data

Module

- Modules are the units of work that Ansible ships out to remote machines.

Ansible refers to the collection of available modules as a library.

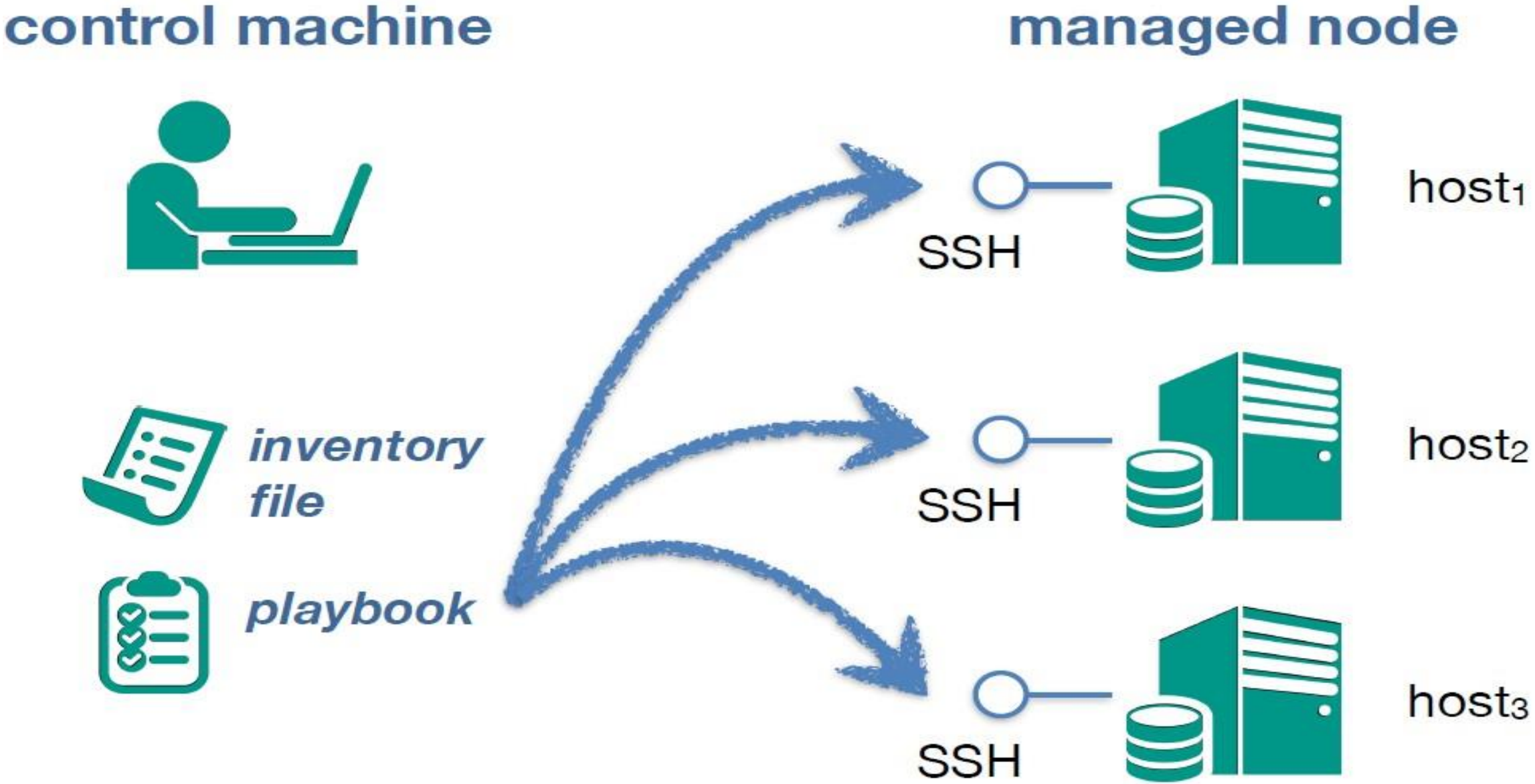
Install Java

Download Oracle JDK

```
get_url:

url:http://download.oracle.com
    dest:jdk-1.8.0-linux-x64.rpm
```

Install Oracle JDK

```
yum:
    name:
jdk-1.8.0-linux-x64.rpm
    state: present
```


control machine
managed node
inventory file
playbook
SSH
SSH
SSH
host1
host2
host3

Playbook Example

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

  tasks:
  - name: ensure apache is at the latest version
    yum:
      name: httpd
      state: latest
  - name: write the apache config file
    template:
      src: /srv/httpd.j2
      dest: /etc/httpd.conf

- hosts: databases
  remote_user: root

  tasks:
  - name: ensure postgresql is at the latest version
    yum:
      name: postgresql
      state: latest
  - name: ensure that postgresql is started
    service:
      name: postgresql
      state: started
```

1
4

Playbook

Plays

Tasks

YAML

Source: Ansible User Guide

Playbook Example 2

```
---
- name: Ensure Apache/PHP is installed.
  apt: "pkg={{ item }} state=latest"
  with_items:
    - apache2
    - php5-common
    - libapache2-mod-php5
    - php5-cli

- name: Ensure Apache is started.
  service: name=apache2 state=started
```

(save as web.yml)

Name of group/host defined in inventory

```
---
- hosts: lamp
  sudo: yes

  tasks:
    - apt: update_cache=yes

    - include: web.yml
```

Include the playbook we created earlier

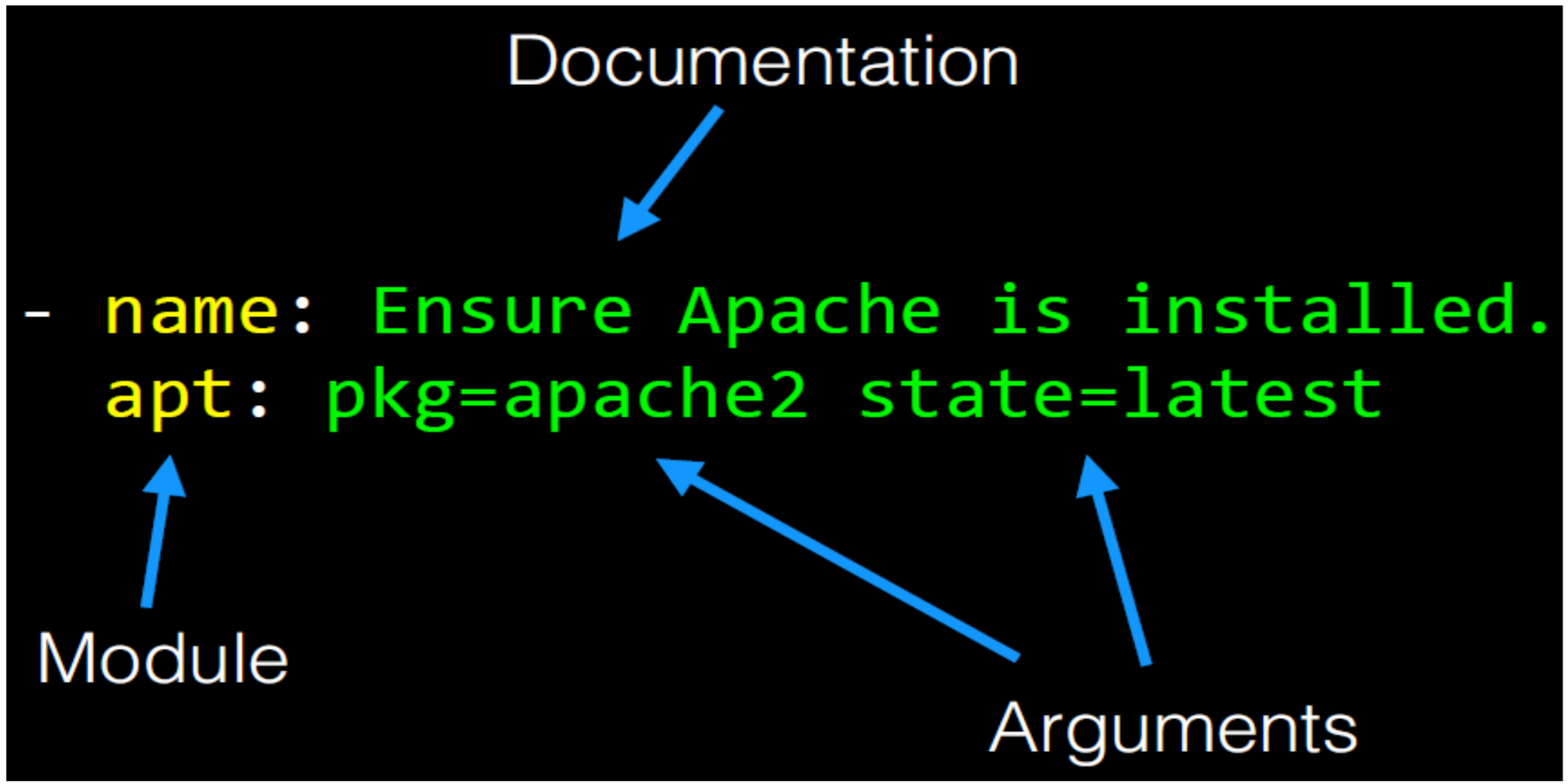
(save as playbook.yml)

Modules

- Modules are API's transferred to the target system and executed to satisfy the task declaration

- apt/yum
- copy
- file
- get_url
- git
- ios_config
- nxos_config
- eos_config
- ping
- debug
- service
- synchronize
- template
- etc.

Playbook Tasks



Playbook Run

```
 09:55 PM $ ansible-playbook playbook.yml

PLAY [lamp] ********************************************************************

GATHERING FACTS ****************************************************************
ok: [192.168.33.3]

TASK: [apt update_cache=yes] ***************************************************
ok: [192.168.33.3]

TASK: [Ensure Apache/PHP is installed.] ****************************************
changed: [192.168.33.3] => (item=apache2,php5-common,libapache2-mod-php5,php5-cli)

TASK: [Ensure Apache is started.] **********************************************
ok: [192.168.33.3]

PLAY RECAP *********************************************************************
192.168.33.3               : ok=4    changed=1    unreachable=0    failed=0
```

Summary

- **Ansible Nodes** There 2 types. Control and Managed. Control is the Ansible server and Managed are devices the Ansible server applies configurations against. Can also manage network devices as well.
- **Inventory** An inventory contains lists or groups of devices that Ansible can run ad-hoc and/or playbooks against. Inventories can be static or dynamic.
- **Ansible Configuration File** Ansible provides default environment settings out of the box. The configuration file can be customized to meet specific needs.
- **Ad Hoc Commands** An Ansible Ad-hoc command is useful for testing tasks without the need to write a full blown playbook. Ad-hoc commands are not persisted.
- **Playbook** An Ansible playbook consists of plays and tasks.
- **Modules** These provide a set of tasks focused on various services and commands.

Configuration File

Ansible Configuration file

- Ansible installation can be customized by modifying settings in the Ansible configuration file
- Using **/etc/ansible/ansible.cfg**
  - Ansible package provides a **base configuration file** located at /etc/ansible/ansible.cfg.
- Using **~/.ansible.cfg**
  - Ansible looks for a ~/.ansible.cfg in the **user's home directory**.
- Using **./ansible.cfg**
  - If an ansible.cfg file exists in the directory in which the ansible command is executed, it is used instead of the global file or the user's personal file.

Managing Ansible Configuration file

- Managing several sections, containing **key-value pairs**
- Enclosed in square brackets ([])
- 6 sections in default ansible configuration file

/etc/ansible/ansible.cfg

```
[defaults]
[privilege_escalation]
[paramiko_connection]
[ssh_connection]
[accelerate]
[selinux]
```

Managing Ansible Configuration file

[default]

- Configuration file grouped under default section.

[privilege_escalation]

- defining how operations that require escalated privileges are executed on managed hosts.

[paramiko_connection], **[ssh_connection]**, and **[accelerate]**

- contain settings for optimizing connections to managed hosts.

[selinux]

- contains settings for defining how SELinux interactions are configured.

[galaxy]

- *Ansible* package provided [galaxy] section. Defining parameters related to Ansible Galaxy,

Configuring Connections

+ How to communicate with its managed hosts?

cat /etc/ansible/ansible.cfg

[defaults]
inventory = ./inventory → The location of the Ansible inventory.
remote_user = <someuser> → Remote user account used to establish connections to managed hosts.
ask_pass = false → Prompt for a password to use when connecting as the remote user.

[privilege_escalation]
become = true → Enable or disable privilege escalation for operations on managed hosts.
become_method = sudo → The privilege escalation method to use on managed hosts.
become_user = root → The user account to escalate privileges to on managed hosts.
become_ask_pass = false → Defines whether privilege escalation on managed hosts should prompt for a password.

Configuration File Comments

Two comment characters are allowed by Ansible configuration files:

- **#** character
  - at the start of a line comments out the entire line. It must not be on the same line with a directive.

- **;** character
  - comments out everything to the right of it on the line. It can be on the same line as a directive, as long as that directive is to its left.

Ansible CLI – Ansible Core Command

- Ansible has a few CLI options allowing for usage without needing a playbook.
- Some of the CLI options we will discuss:
  - Ansible
  - Ansible-doc
  - Ansible-config
  - Ansible-console
- There are other options that are not as popular, check the command_line_tools section of the documentation.

Ansible Core Command

ansible –list-hosts “group/host”

```
#ansible --list-hosts all
  hosts (3):
    veos-pod-00.localdomain
    localhost
    csr1000v-pod-00.localdomain

# ansible --list-hosts switches
  hosts (1):
    veos-pod-00.localdomain

# ansible --list-hosts routers
  hosts (1):
    csr1000v-pod-00.localdomain

# ansible --list-hosts veos*
  hosts (1):
    veos-pod-00.localdomain
```

Ansible Core – Passwords/SSH Keys

- So we have a few options with Passwords and SSH with the Ansible core command:
  - -k, --ask-pass – Tells ansible to ask for a connection password
  - -u, --user – Connect as this user, by default this will not be set.
  - --private-key, --key-file – Use this key file to auth the connection
- Host Key Checking
  - Enabled by default in Ansible
  - Can be disabled by modifying ansible.cfg

```
Ansible.cfg

[defaults]
Host_key_checking = False
```

Ansible-doc

+ Used to allow searching the documentation via the CLI
+ ansible-doc -l, --list – Will list all available plugins, supports regex to filter:

```
# ansible-doc -l | grep ^nxos
nxos_aaa_server
Manages AAA server global configuration.
nxos_aaa_server_host
Manages AAA server host-specific configuration.
nxos_acl
Manages access list entries for ACLs.
nxos_acl_interface
Manages applying ACLs to interfaces.

…More continued
```

Ansible-doc cont…

- Can also use the –s option to show a snippet of the command for the playbook.
- ansible-doc -s, --snippet

```
- name: Manages VLAN resources and attributes.
  nxos_vlan:
      admin_state:           # Manage the VLAN administrative state of the VLAN equivalent to shut/no shut in VLAN config mode.
      aggregate:             # List of VLANs definitions.
      associated_interfaces:   # This is a intent option and checks the operational state of the for given vlan `name' for
                               associated interfaces. If the value in the
                               `associated_interfaces' does not match with the operational state
                               of vlan interfaces on device it will result in failure.
      delay:                 # Time in seconds to wait before checking for the operational state on remote device. This wait is
                               applicable for operational state arguments.
      interfaces:            # List of interfaces that should be associated to the VLAN or keyword 'default'.
      mapped_vni:            # The Virtual Network Identifier (VNI) ID that is mapped to the VLAN. Valid values are integer and
                               keyword 'default'. Range 4096-16773119.
      mode:                  # Set VLAN mode to classical ethernet or fabricpath. This is a valid option for Nexus 5000 and
                               7000 series.
      name:                  # Name of VLAN or keyword 'default'.
      provider:              # *Deprecated* Starting with Ansible 2.5 we recommend using `connection: network_cli'. This option
                               is only required if you are using NX-API. For more information
                               please see the L(NXOS Platform Options guide,
                               ../network/user_guide/platform_nxos.html). HORIZONTALLINE A dict
                               object containing connection details.
      purge:                 # Purge VLANs not defined in the `aggregate' parameter. This parameter can be used without
                               aggregate as well.
      state:                 # Manage the state of the resource.
      vlan_id:               # Single VLAN ID.
      vlan_range:            # Range of VLANs such as 2-10 or 2,5,10-15, etc.
      vlan_state:            # Manage the vlan operational state of the VLAN
```

Ansible-config

- Allows for viewing, editing and managing the ansible configuration.

```
# ansible-config view
[defaults]
host_key_checking = False
inventory = ./inventory
ansible_local_temp = /tmp
```

- Can also be used to see the differences in the config against the original defaults:

```
# ansible-config dump --only-changed

DEFAULT_HOST_LIST(/root/ansible_lab_files/lab1-cli-
tools/ansible.cfg) = [u'/root/ansible_lab_files/lab1-cli-
tools/inventory']
HOST_KEY_CHECKING(/root/ansible_lab_files/lab1-cli-
tools/ansible.cfg) = False
```

Ansible-console

- Used as a REPL (Read-Evaluate-Print-Loop) for executing ansible tasks interactively.
- Can be specified to use a certain group or host
- Has all the same cmd line options like –u for username and –k for password

Ansible-console cont…

```
# ansible-console server -u root -k
SSH password:
Welcome to the ansible console.
Type help or ? to list commands.

root@server (1)[f:5]$ ping
localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"}
root@server (1)[f:5]$ pwd
localhost | SUCCESS | rc=0 >>
/root

root@server (1)[f:5]$ ifconfig -a
localhost | SUCCESS | rc=0 >>
ens160: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.16.15.200  netmask 255.255.0.0  broadcast 172.16.255.255
        inet6 fe80::365f:1f69:6746:75e8  prefixlen 64  scopeid 0x20<link>
        ether 00:50:56:b7:a3:c6 txqueuelen 1000  (Ethernet)
        RX packets 9572571  bytes 891289656 (850.0 MiB)
        RX errors 0  dropped 10699  overruns 0  frame 0
        TX packets 63640 bytes 5607494 (5.3 MiB)
        TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Ansible-console cont…

```
# ansible-console nxos -u admin -k  SSH password:
Welcome to the ansible console.  Type help or ? to list commands.

admin@nxos (1)[f:5]$ raw show version | grep "NXOS: version"
n9k-standalone-01.localdomain | SUCCESS | rc=0 >>  NXOS: version 7.0(3)I2(1)
Shared connection to n9k-standalone-01.localdomain closed.

admin@nxos (1)[f:5]$ raw show logging logfile | last 10
n9k-standalone-01.localdomain | SUCCESS | rc=0 >>
2018 Jul 28 01:25:21 N9k-Standalone-Pod-1 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configured from vty by admin on 192.168.100.10@pts/1  2018 Jul 28 01:
message repeated 1 time
2018 Jul 28 01:39:59 N9k-Standalone-Pod-1 last message repeated 5 times
2018 Jul 28 01:39:59 N9k-Standalone-Pod-1 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configured from vty by admin on 192.168.100.10@pts/0  2018 Jul 31 10
VSHD_SYSLOG_CONFIG_I: Configured from vty by admin on 172.16.200.100@pts/0  2018 Jul 31 10:23:15 N9k-Standalone-Pod-1 last message repeated 1
2018 Aug 14 18:58:03 N9k-Standalone-Pod-1 %AUTHPRIV-3-SYSTEM_MSG: pam_aaa:Authentication failed for user admin from 172.16.15.200 -  sshd[262
2018 Aug 14 18:58:03 N9k-Standalone-Pod-1 %DAEMON-3-SYSTEM_MSG: error: PAM: Authentication failure for admin from 172.16.15.200 -  sshd[26217
2018 Aug 14 19:14:14 N9k-Standalone-Pod-1 %VSHD-5-VSHD_SYSLOG_CONFIG_I: Configured from vty by admin on 10.1.1.22@pts/1  2018 Aug 14 19:27:33
message repeated 1 time
Shared connection to n9k-standalone-01.localdomain closed.
```

Ad-hoc Commands

- Allows us to run one off commands against 1 or multiple devices by using hosts or groups.
- Can be extremely useful in troubleshooting or when looking up information quickly.

Ad-Hoc Commands on Local Machine

Ping the localhost

```
$ ansible -m ping localhost
 [WARNING]: provided hosts list is empty, only localhost is available


localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

Ad-Hoc Commands on Inventory

Ping the hosts and groups you defined in Inventory

```
$ ansible -m ping web1
$ ansible -m ping app
$ ansible -m ping webservers
$ ansible -m ping dc
```

Ad-hoc Commands more examples

```
# ansible nxos -m raw -a "show cdp neighbors" -u admin -k
password:
n9k-standalone-01.localdomain | SUCCESS | rc=0 >>
Capability Codes: R - Router, T - Trans-Bridge, B - Source-Route-Bridge  S - Switch, H - Host, I - IGMP, r - Repeater,
                  V - VoIP-Phone, D - Remotely-Managed-Device,  s - Supports-STP-Dispute

Device-ID          Local Intrfce  Hldtme Capability  Platform       Port ID

OC-3750E-3.onecloud.com
                    mgmt0          133    R S I      WS-C3750E-24T Gig1/0/5
N5k-1(SSI15290E2B)
                    Eth1/11        151    R S I s N5K-C5548UP  Eth1/22
N9k-Spine-1(SAL1833YM38)
                    Eth1/52        171    R S s      N9K-C9396PX   Eth2/2
N9k-Spine-2(SAL1832Y6XS)
                    Eth1/53        171    R S s      N9K-C9396PX   Eth2/2

Total entries displayed: 4
```

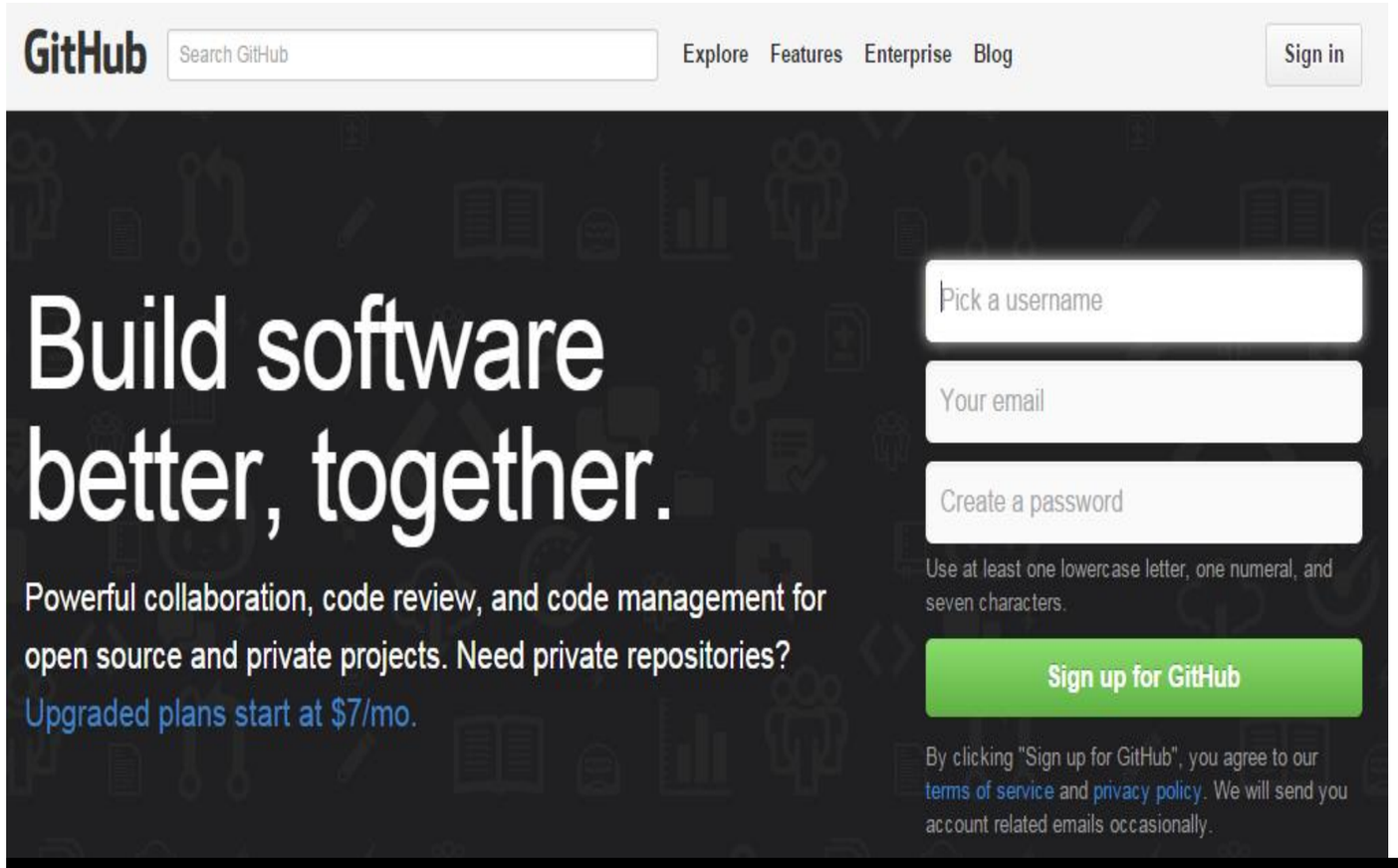

git

THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.

COOL. HOW DO WE USE IT?

NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

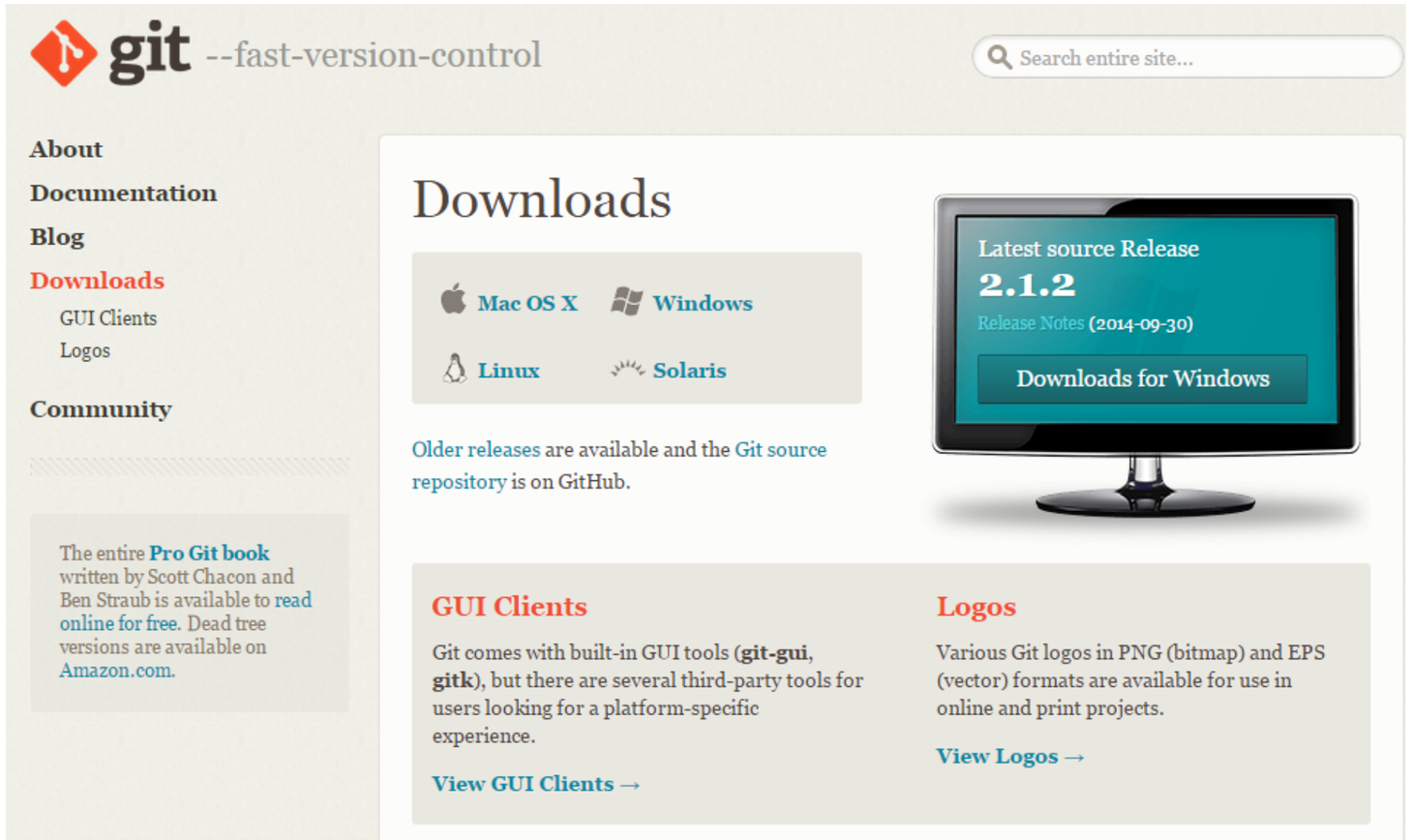
What's GitHub?



- **Web-based repository**
- **GUI driven**
- Distributed version control
- Source code management
- Host software projects
- Used with Git

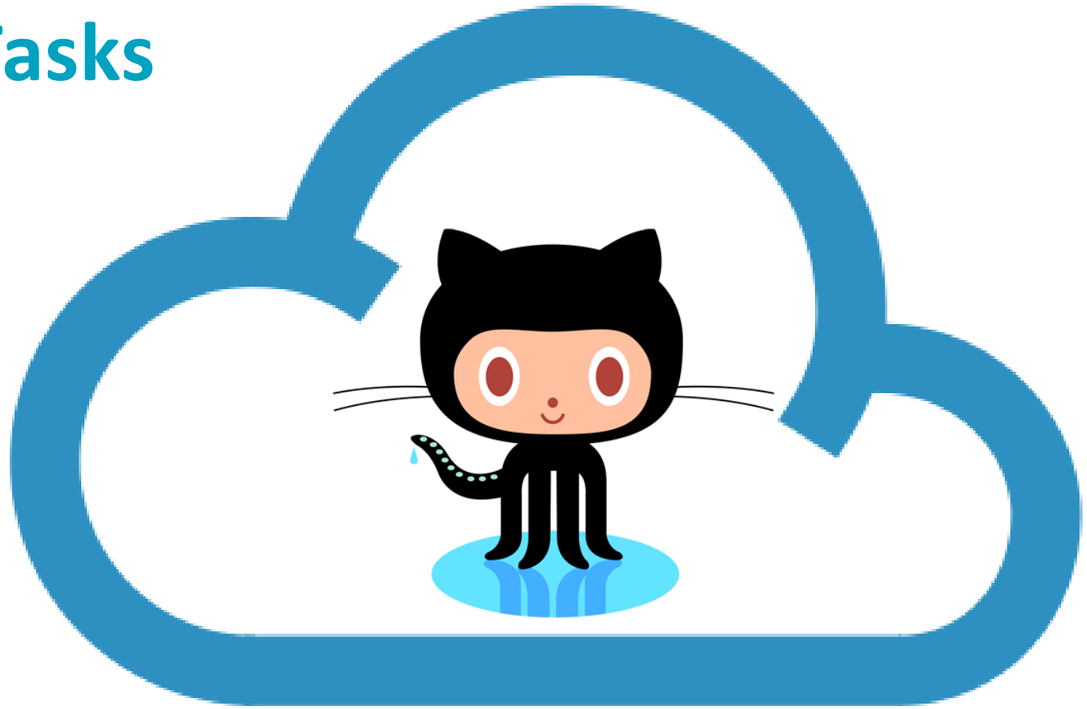
http://github.com

What's Git?



http://git-scm.com

Common GitHub Tasks

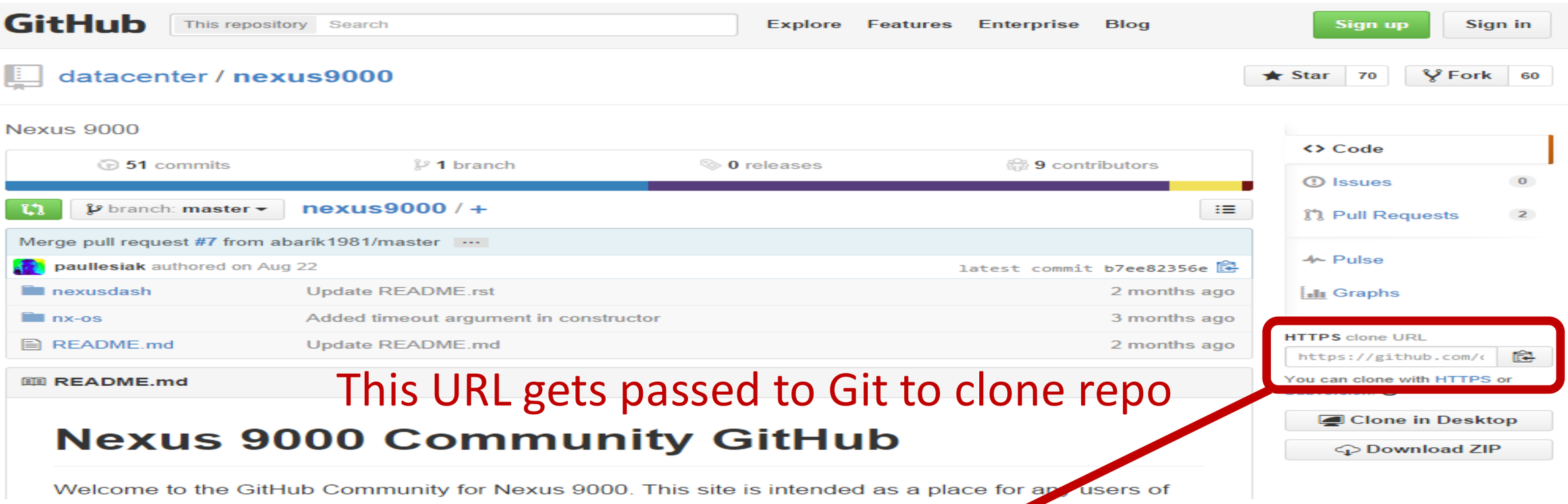


Pull

Push

Using Git “PULL” to Clone an Existing GitHub Repository

http://github.com



Git Command Line	Description
git clone https://github.com/datacenter/nexus9000.git	Pulls Nexus9K GitHub repository to local Git repository

“Hello World” Using Git “PUSH”

Setting up local Git repository

Step	Example
Create a project	mkdir hello-world cd hello-world
Initialize repository	git init
Add new source files at this step	
Track files	1) git add . 2) git add file.filetype 3) git add dir
Commit all changes	git commit –m “commit message”
Push files t*o* remote GithHub repo	git push –u origin master

Lab 0

GIT

Lab 1

CLI Tools & Tricks

Lab 2

Adhoc-commands

1

6

OneCloud Consulting

The Cloud Services Division of ePlus

Ansible Troubleshooting & Tools

Module 5

Objectives

+ logging
+ debug module
+ error management and debugging
+ verbosity
+ test modules
+ Linting tools

Logging

Log Files in Ansible

- Ansible logging is **NOT** enabled by default
- Ansible provides logging through the **log_path** parameter
- Logging is enabled in the **default** section of the **ansible.cfg** configuration file using the **$ANSIBLE_LOG_PATH** environment variable
- If Ansible logs are stored in the default log file directory, **/var/log**, playbooks **MUST** be run as **root** or permissions on **/var/log** must be opened up

Debug Module

Use Cases

+ Can be used for verifying tasks that register variables have set the variable properly.

+ You could use it for loops and such to see what information is being looped over.

+ Can be used to get the return values/codes of a script or call you make.

Example Use Case

- We need to set a interface MTU to 9216 from 1500 and verify the change actually occurs.
- We will use the debug module to see our results of our facts after the MTU change task runs.
- Other use cases might be checking ansible server resources or logs or debug just playbook output in general for when you are building out new playbooks.

Example Use Case Playbook

```
---

- name: Change MTU Setting
  hosts: nxos
  connection: local
  gather_facts: yes

  tasks:
    - nxos_facts:
    - name: Current MTU setting
      debug: var=ansible_net_interfaces['Ethernet1/1']['mtu']
    - name: Change MTU to 9000
      nxos_interface:
        name: Ethernet1/1
        mtu: 9216
    - nxos_facts:
    - debug: var=ansible_net_interfaces['Ethernet1/1']['mtu']
```

Example Use Case Output

```
ansible-playbook debug_example.yaml -u admin -k
SSH password:

PLAY [Change MTU Setting] ******************************************************************************************************************

TASK [Gathering Facts] *********************************************************************************************************************
ok: [n9k-standalone-01.localdomain]

TASK [nxos_facts] **************************************************************************************************************************
ok: [n9k-standalone-01.localdomain]

TASK [Current MTU setting] *****************************************************************************************************************
ok: [n9k-standalone-01.localdomain] => {
    "ansible_net_interfaces['Ethernet1/1']['mtu']": "1500"
}

TASK [Change MTU to 9000] ******************************************************************************************************************
changed: [n9k-standalone-01.localdomain]

TASK [nxos_facts] **************************************************************************************************************************
ok: [n9k-standalone-01.localdomain]

TASK [debug] *******************************************************************************************************************************
ok: [n9k-standalone-01.localdomain] => {
    "ansible_net_interfaces['Ethernet1/1']['mtu']": "9216"
}

PLAY RECAP *********************************************************************************************************************************
n9k-standalone-01.localdomain : ok=6 changed=1 unreachable=0 failed=0
```

Error Management and Debugging

About Errors

What You Need to Know

- Several issues than can occur during a playbook run, mainly related:
  - Syntax error
  - Template error
  - Connectivity error (managed hosts in the inventory file)
- Errors are issued by the **ansible-playbook** command at execution time

What You Can Do

Use the **--syntax-check** option to check the YAML syntax of the playbook

You can also use the **--step** or **--start-at-task** options to step through tasks or test specific tasks in a playbook

Example: Debugging with ansible-playbook

- Output given by a playbook run with the **ansible-playbook** command
- Good starting point to start troubleshooting issues related to hosts managed by Ansible

```
PLAY [playbook]
*********************************************************************
... Output omitted ...
TASK: [Install a service]
*********************************************************************
ok: [demoservera]
ok: [demoserverb]
PLAY RECAP
*********************************************************************
demoservera : ok=2 changed=0 unreachable=0 failed=0
demoserverb : ok=2 changed=0 unreachable=0 failed=0
```

More about Debugging with ansible-playbook

```
PLAY [playbook]
****************************************************
***************
... Output omitted ...
TASK: [Install a service]
****************************************************
ok: [demoservera]
ok: [demoserverb]
PLAY RECAP
****************************************************
*****************                                    1
demoservera : ok=2 changed=0 unreachable=0 failed=0
demoserverb : ok=2 changed=0 unreachable=0 failed=0    3
```

- Output shows a **PLAY** header with the name of the play to be executed
- One or more **TASK** headers are included
- Each of these headers represent their associated ***task*** in the playbook
- Executed in all the managed hosts belonging to the **group** included in the playbook in the ***hosts*** parameter
- Each managed host executes the tasks, which can be set to **ok**, **fatal**, or **changed**.
- **PLAY RECAP** section shows the number of tasks executed for each managed host as its output state.

Example: ansible-playbook –start-at-task

The **--start-at-task** option starts the execution from a given task and avoids repeating previous task executions.

```
ansible-playbook play.yml -step

ansible-playbook play.yml --start-at-task="start httpd service"

ansible-playbook play.yml --syntax-check
```

Verbosity

1

5

Verbosity Configuration

- **ansible-playbook –v** command provides additional debugging information, with up to four levels of verbosity.

Verbosity configuration	
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	Adds extra verbosity options to the connection plug-ins, including the users being used in the managed hosts to execute scripts, and what scripts have been executed.

Test Modules

Modules for Testing: assert

```
- name: verify nxos version
  hosts: nxos
  connection: local
  gather_facts: yes

  tasks:
    - nxos_facts:
    - name: Current NXOS Version
      debug: var=ansible_net_version
    - name: Verify Version is 7.x
      assert:
        that:
          - "'{{ ansible_net_version }}' is match('^7')"
```

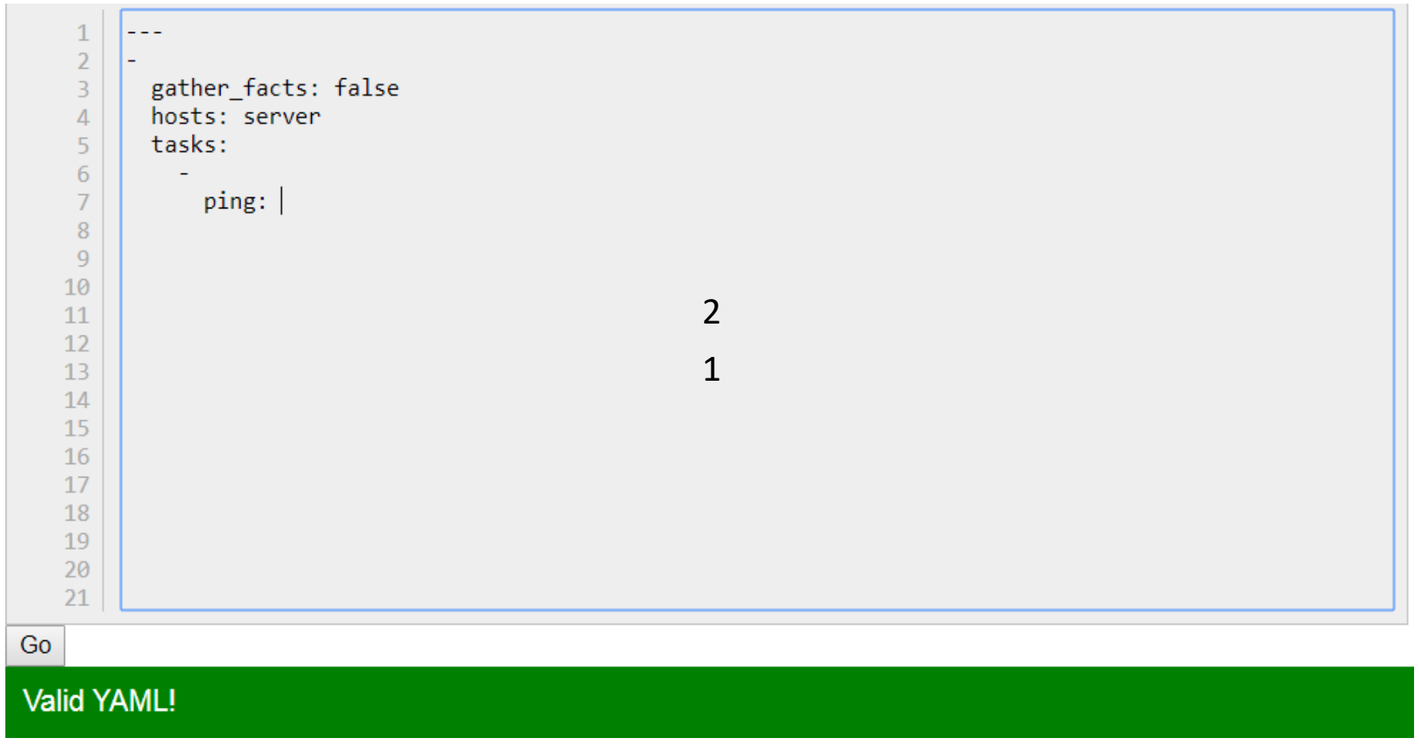
Using ad hoc Commands for Testing

```
ansible node1* -u devops -b -m yum -a 'name=httpd state=present‘

ansible node1* -a 'lsblk‘

ansible node1* -a 'free -m‘

ansible switches -m raw -a "show version" -u admin -k

ansible switches -m raw -a "show logging log | last 10" -u admin -k
```

Linters & Tools

YAML Linter

+ www.yamllint.com – A web based YAML linter for checking syntax



Lint, or a **linter**, is a tool that analyzes source code to flag programming errors, bugs, stylistic errors, and suspicious constructs. The term originates from a Unix utility that examined C language source code.

Ansible Linter

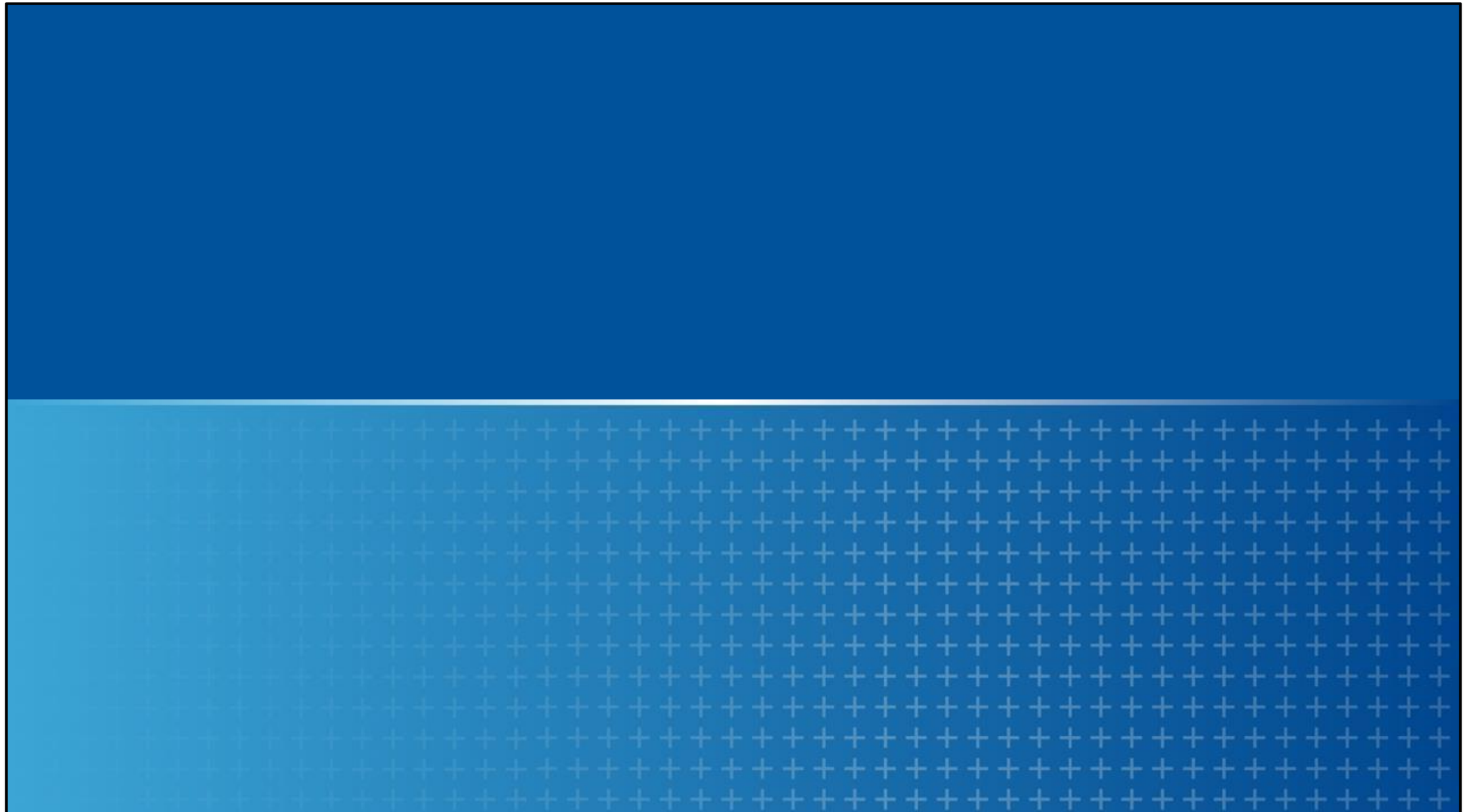
- https://github.com/willthames/ansible-lint
- Can be installed easily via pip – pip install ansible-lint

```
# ansible-lint ping_server.yaml
[ANSIBLE0011] All tasks should be named
ping_server.yaml:5
Task/Handler: ping
```

```
# cat ping_server.yaml
---
- hosts: server
  gather_facts: true
  tasks:
    - ping:
```

Lab 3

Troubleshooting & Tools



OneCloud
Consulting
The Cloud Services Division of ePlus

Inventory & Playbooks

Module 6

Objectives

- Directory Structure
- Ansible inventory
- Dynamic inventory
- Playbook configuration
- Playbook examples
- What are playbooks used for

Directory Structure

Directory Structure - Overview

```
production              # inventory file for production devices
staging                 # inventory file for staging environment


group_vars/
   group1.yml           # here we assign variables to particular groups
   group2.yml

host_vars/
   hostname1.yml        # here we assign variables to particular systems
   hostname2.yml
```

Directory Structure – Overview Cont.

```
library/                # if any custom modules, put them here (optional)
module_utils/           # if any custom module_utils to support modules, put them here (optional)
filter_plugins/         # if any custom filter plugins, put them here (optional)


site.yml                # master playbook
switches.yml            # playbook for switches tier
routers.yml             # playbook for routers tier
```

Directory Structure – Overview Cont.

```
roles/
    common/               # this hierarchy represents a "role"
        tasks/            #
            main.yml # <-- tasks file can include smaller files if warranted
        handlers/         #
            main.yml # <-- handlers file
        templates/ # <-- files for use with the template resource
            ntp.conf.j2 # <------- templates end in .j2
        vars/             #
            main.yml # <-- variables associated with this role
        defaults/         #
            main.yml # <-- default lower priority variables for this role
        meta/             #
            main.yml # <-- role dependencies
```

Directory Structure – Alternative Option

- We just showed from the Ansible documentation what they have listed as the best practice for the directory layout.

- There is also an alternate layout they recommend that requires more files but consists of putting each inventory file and group_vars for it in a separate dir.

Directory Structure – Alternative Option Cont.

```
inventories/
   production/
      hosts              # inventory file for production devices
      group_vars/
         group1.yml      # here we assign variables to particular groups
         group2.yml
   host_vars/
      hostname1.yml    # here we assign variables to particular devices
      hostname2.yml
```

Directory Structure – Alternative Option Cont.

Then our roles would just look something like:

roles/

- common/
- update_vlans/
- ntp/

This layout gives some more options and flexibility but at the cost of more files. Typically would be better for larger environments.

Roles will be discussed later

1

0

0

Inventory

Ansible Inventory

- Collection of **hosts**
- Used for managing multiple servers in the infrastructure **/etc/ansible/hosts**
- Hosts can also be assigned to ***groups***
- **Groups** can contain ***child*** groups, and **hosts** can be members of ***multiple*** groups.
- The inventory can set ***variables*** that apply to the ***hosts*** and ***groups***
- **Static Inventory**
  - ***Text file*** - specifies the managed hosts
  - List of *host names* or *IP addresses*
- **Dynamic Inventory**
  - Dynamically generated – scripted, external source

1

Static Inventory Example

web1.example.com → hostname

web2.example.com

db1.example.com

db2.example.com

192.0.2.42 → IP Address

Static Inventory with host and groups Example

[master] → host_group1 / parent1

podx-master.origin.com — hostname

[nodes] → host_group2 / parent2

podx-node1.origin.com

podx-node2.origin.com

172.1.120.xx

[everyone:children] → host_group3 / children1

master

nodes

Inventory host groups

Two host groups always exist:

- The **all** host group contains every host explicitly listed in the inventory.
- The **ungrouped** host group contains every host explicitly listed in the inventory that isn't a member of any other group.

Defining Nested Groups

- Ansible host inventories can include groups of host groups.
- This is accomplished with the **:children** suffix.

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

[canada]
ontario01.example.com

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

Simplifying Host Specifications with Ranges

- Ansible host inventories can be simplified by specifying ranges in the host names or IP addresses.
- Numeric ranges can be specified, but alphabetic ranges are also supported.
- Ranges have the following syntax:


```
[START:END]
```

Patterns Examples

Decides which hosts to manage

- All hosts in the inventory (all or *)
- A specific host name or group name (host1, webservers)
- Wildcard configuration (192.168.1.*)
- OR configuration (host1:host2, webservers:dbservers)
- NOT configuration (webservers:dbservers:!production)
- AND configuration (webservers:dbservers:&staging)
- REGEX configuration (~(web|db).*\.example\.com)
- Exclude hosts using limit flag (ansible-playbook site.yml --limit datacenter2)

Ranges match all the values from *START* to *END*, inclusive.

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

Testing the Inventory

Test the machine's *presence* in the inventory with the **ansible** command:

```
ansible washington1.example.com --list-hosts

hosts (1):
washington1.example.com
```

```
ansible washington01.example.com --list-hosts

hosts (0):
```

Dynamic Inventory

- Ansible inventory information can also be dynamically generated, using information provided by external databases.
- The open source community has written a number of dynamic inventory scripts that are available from the upstream Ansible project

Dynamic Inventory[1][2]

Dynamic Inventory

- Dynamic inventory scripts are used just like static inventory text files
- Ansible supports *dynamic inventory* scripts that retrieve current information
- These scripts are executable programs that collect information from some external source and output the inventory in JSON format.
- The location of the inventory is specified either directly in the current **ansible.cfg** file or using the **ansible-playbook -i** option

Open Source Dynamic Inventory Scripts

- A number of existing dynamic inventory scripts have been contributed to the Ansible project by the open source community
  - Private cloud platforms, such as Red Hat OpenStack Platform
  - Public cloud platforms, such as Rackspace Cloud, Amazon Web Services EC2, or Google Compute Engine
  - Virtualization platforms, such as Red Hat Virtualization (oVirt) and VMware vSphere.
  - Platform-as-a-Service solutions, such as OpenShift Container Platform
  - Life cycle management tools, such as Foreman (with Red Hat Satellite 6 or stand-alone) and Spacewalk (upstream of Red Hat Satellite 5).
  - Hosting providers, such as Digital Ocean or Linode.

Developing Dynamic Inventory Sources

+ Custom dynamic inventory program
+ It can be written in any programming language, and must return inventory information in JSON format when passed appropriate options.
+ When passed the **--list** option, the script must output a JSON-encoded hash/dictionary of all of the hosts and groups in the inventory to standard output.

```
./inventoryscript --list
```

Managing Multiple Inventories

- Ansible supports the use of multiple inventories in the same run
- **-i** option or the value of the **inventory** parameter in the configuration file is a directory
- All inventory files included in the directory, either static or dynamic, are combined to determine the inventory.
- The executable files within that directory are used to retrieve dynamic inventories, and the other files are used as static inventories.
- Inventory files should not depend on other inventory files or scripts in order to resolve.

Playbooks

Hosts

- A **hosts** file consists of **host groups** and **hosts** within those groups.
- A super-set of hosts can be built from other host groups using the **:children** operator.
- These files can be found within a Ansible **repository** or in **/etc/ansible/hosts**

Host Selector

```
---
- name: install and start apache
  hosts: web
  become: yes
  vars:
    http_port: 80
  tasks:
     - name: httpd package is present
       yum:
         name: httpd
         state: latest
   - name: latest index.html file is present
     copy:
       src: files/index.html
       dest: /var/www/html/
```

Host name

Tasks

Tasks are the application of a **module** to perform a specific unit of work.

- **file**: A directory should exist
- **yum**: A package should be installed
- **service**: A service should be running
- **template**: Render a configuration file from a template
- **get_url**: Fetch an archive file from a URL
- **git**: Clone a source code repository

Example of Tasks in a Play

```
tasks:
- name: httpd package is present
- yum:
    name: httpd  state: latest

- name: latest index.html file is present
- copy:
    src: files/index.html
    dest: /var/www/html/

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

Handler Tasks

Handlers are special tasks that run at the end of a play if notified by another task when a change occurs.

```
tasks:
  - name: Configure Nexus VLANs
    nxos_vlan:
      vlan_id: "{{ item.vlan_id }}"
      admin_state: "{{ item.admin_state }}"
      name: "{{ item.name }}"
    with_items: "{{ vlans }}"
    notify: write mem

handlers
  - name: write mem
    nxos_command:
      commands: copy run start
    tags: wr_mem
```

Plays and Playbooks

- Plays are ordered sets of tasks to execute against host selections from your inventory.
- A playbook is a file containing one or more plays.
- Expressed in YAML language
- Composed of one or more “plays” in a list
- Allowing multi-machine deployments orchestration

Playbook Example

```
---

- name: Playbook to configure SNMP community string.
  hosts: nxos
  gather_facts: yes

  tasks:
    - name: Configuring SNMP community
      nxos_snmp_community:
        community: mycommunity
        group: network-operator
        state: absent
```

3
3

Playbook - tasks

- Are executed in order against all machines matched by the host pattern
- May be Included from other files

```
tasks:
- include: tasks/foo.yml
```

- Hosts with failed tasks are taken out for the entire playbook
- Each task executes a module with specific options
- Modules are idempotent in order to bring the system to the desired state

tasks:

```
- name: {task name}
        {module}: {options}
```

Playbook - handlers

- Notifications may be triggered at the end of each block of tasks whenever a change has been made on the remote system
- Handlers are referenced by name

```
tasks:
  - name: Configure Nexus VLANs
    nxos_vlan:
      vlan_id: "{{ item.vlan_id }}"
      admin_state: "{{ item.admin_state }}"
      name: "{{ item.name }}"
    with_items: "{{ vlans }}"
    notify: write mem

handlers:
  - name: write mem
    nxos_command:
      commands: copy run start
    tags: wr_mem
```

Playbook – Privilege Escalation

- There is a option we can use in our playbooks or group vars called become, this replaces the old authorize and auth_pass options.
- When using with network devices it requires the connection: network_cli or connection: httpapi to be able to use become.
- Become is used for privilege escalation on network devices or also known as enable on the actual switches.

Playbook – Privilege Escalation Cont.

- We can use escalated privileges on specific tasks, an entire play, or even all plays.
- Add become at the tasks level for a specific task:

```
- name: Gather facts (eos)
  eos_facts:
  become: yes
  become_method: enable
```

Playbook – Privilege Escalation Cont.

- We can also set enable mode for all tasks in a single play:

```
- hosts: eos-switches
  become: yes
  become_method: enable
  tasks:
    - name: Gather facts (eos)
      eos_facts:
```

Playbook – Privilege Escalation Cont.

- Finally we can set it up so all tasks in all plays use privilege mode. We can achieve this using group_vars

```
Group_vars/eos.yml

ansible_connection: network_cli
ansible_network_os: eos
ansible_user: myuser
ansible_become: yes
ansible_become_method: enable
```

Playbook – Privilege Escalation Cont.

- If your enable mode requires a password you can specify it in 1 of 2 different ways:
  - You can provide the `--ask-become-pass` which will ask for the pw.
  - Set the `ansible_become_pass` connection variable in your vault would be preferred.

Vault

Ansible Vault

- Allows us to store secrets or passwords securely.
- By default uses PyCrypto to encrypt the files, also supports package Cryptography
- Allows storing of passwords, files, and other options.

Ansible Vault - Format

- The vault encrypted file is formatted in a UTF-8 encoded text file.
- At the top of the file is a terminated header similar to:
  - $ANSIBLE_VAULT;1.1;AES256
  - Vault format id, vault format version, cipher id each separated by ;

```
$ANSIBLE_VAULT;1.1;AES256
66343662363739366330326438613434373830343037363139613563323565303165303865613236
34653134333338363231373163373061623033393830386330a653135343666346666366437646437
32376337336536356534643639386661613533306338356237373233626536663235306563333236
32643733616164323002a31633563626633383162663363653038613935636437313161363666643964366565636432633166373531363138386231323166623939303236613161643462346431393962313365633362623339326561623637653232393061386665303039
```

Ansible Vault – What Can We Encrypt?

- It can encrypt any structured data file that is used by Ansible. Can include the “group_vars” or “host_vars” variables.
- Also can include variables loaded by include_vars or var_files as well as any passed via the ansible-playbook cli with –e
- Can also encrypt tasks, and handlers to hide the names of variables that you might be using.
- As of 2.3 now supports !vault tag to encrypt single values directly inside the YAML file.

Ansible Vault – The Commands

- We can create encrypted files via the “ansible-vault create password.yaml” command.
- Can modify our files via the “ansible-vault edit password.yaml” command
- We can change our ansible vault pw using “ansible-vault rekey password.yaml”
- Also allows us to encrypt a non-encrypted file, “ansible-vault encrypt plaintext_pw.txt”

Summary

- Ansible playbooks are written in YAML format.
- YAML files are structured using space indentation to represent data hierarchy.
- A *playbook* is a text file that contains a list of one or more plays to run in order.
- A *play* is an ordered list of tasks, which should be run against hosts selected from the inventory.
- Tasks are implemented using standardized code packaged as Ansible *modules*.
- The **ansible-playbook** command is used to verify playbook syntax and run playbooks.
- The **ansible-doc** command can list installed modules, and provide documentation and example code snippets of how to use them in playbooks.

Summary (continued)

- Managed hosts are defined in the ***inventory***.
- **Host patterns** are used to reference managed hosts defined in an inventory.
- Inventories can be a **static file** or **dynamically generated** by a program from an external source, such as a directory service or cloud management system.
- The location of the inventory is controlled by the Ansible **configuration file** in use, but most frequently is kept with the playbook files.

Summary (continued)

- Ansible looks for its configuration file in a number of places in order of precedence.
- The first configuration file found is used; all others are ignored.
- The **ansible** command is used to perform *ad-hoc commands* on managed hosts.

- Ad-hoc commands determine the operation to perform through the use of *modules* and their arguments.
- Ad-hoc commands requiring additional permissions can make use of Ansible's *privilege escalation* features.

Lab 4
Inventory

Lab 5

Playbooks

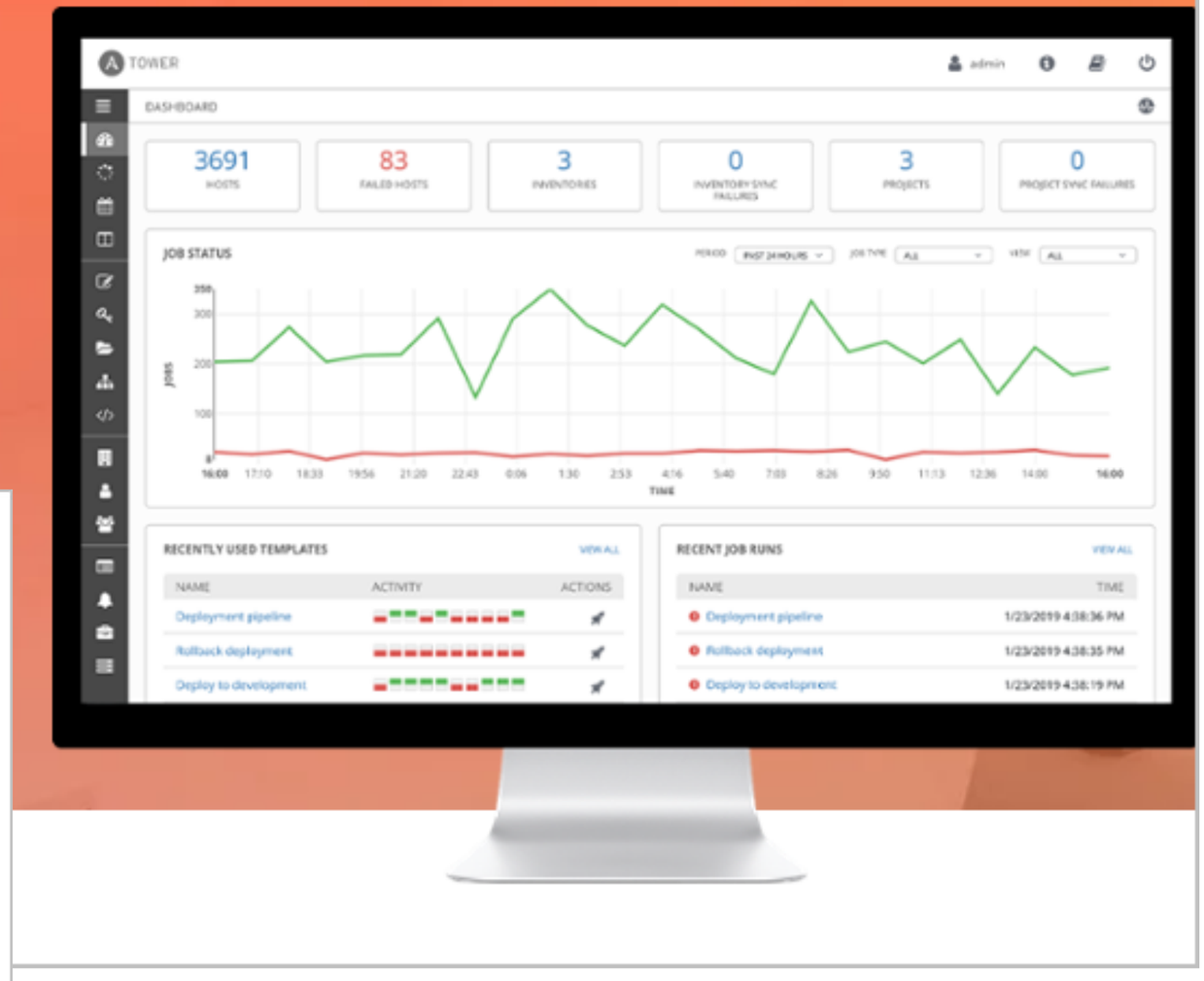
Lab 6

Vault

Ansible Tower

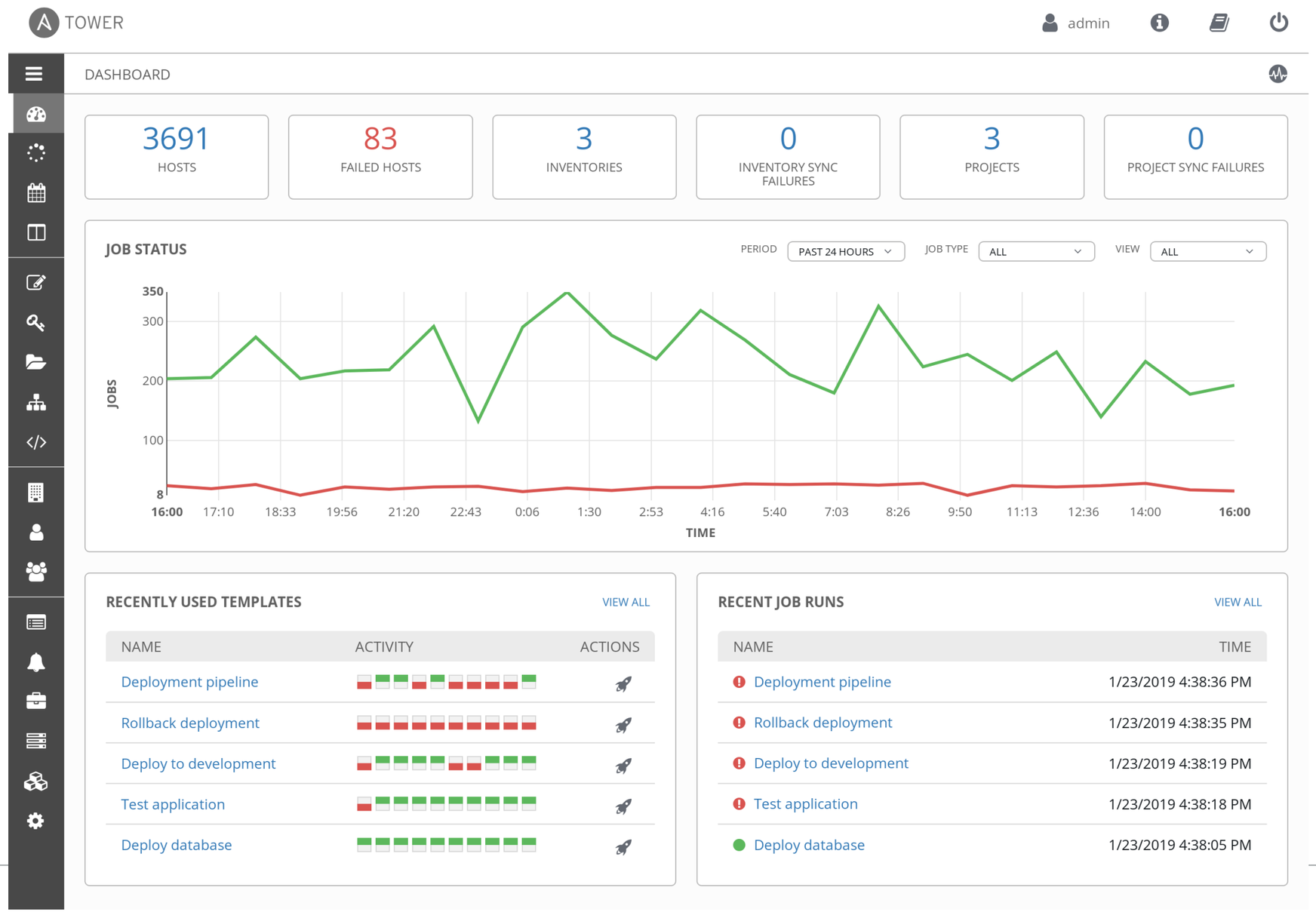
RED HAT®
ANSIBLE®
Tower

SOLVE IT. **AUTOMATE IT.** SHARE IT.



RED HAT®
ANSIBLE®
Tower

YOUR ANSIBLE DASHBOARD



REAL-TIME JOB STATUS UPDATES

TOWER

admin

JOBS / 29170 - Deploy application

DETAILS

STATUS	Successful
STARTED	1/23/2019 6:52:19 PM
FINISHED	1/23/2019 6:52:35 PM
JOB TEMPLATE	Deploy application
JOB TYPE	Run
LAUNCHED BY	admin
INVENTORY	Development Cloud
PROJECT	App deployment
PLAYBOOK	deploy-application.yml
LIMIT	us-east-2b
INSTANCE GROUP	tower

EXTRA VARIABLES YAML JSON EXPAND

```
---
```

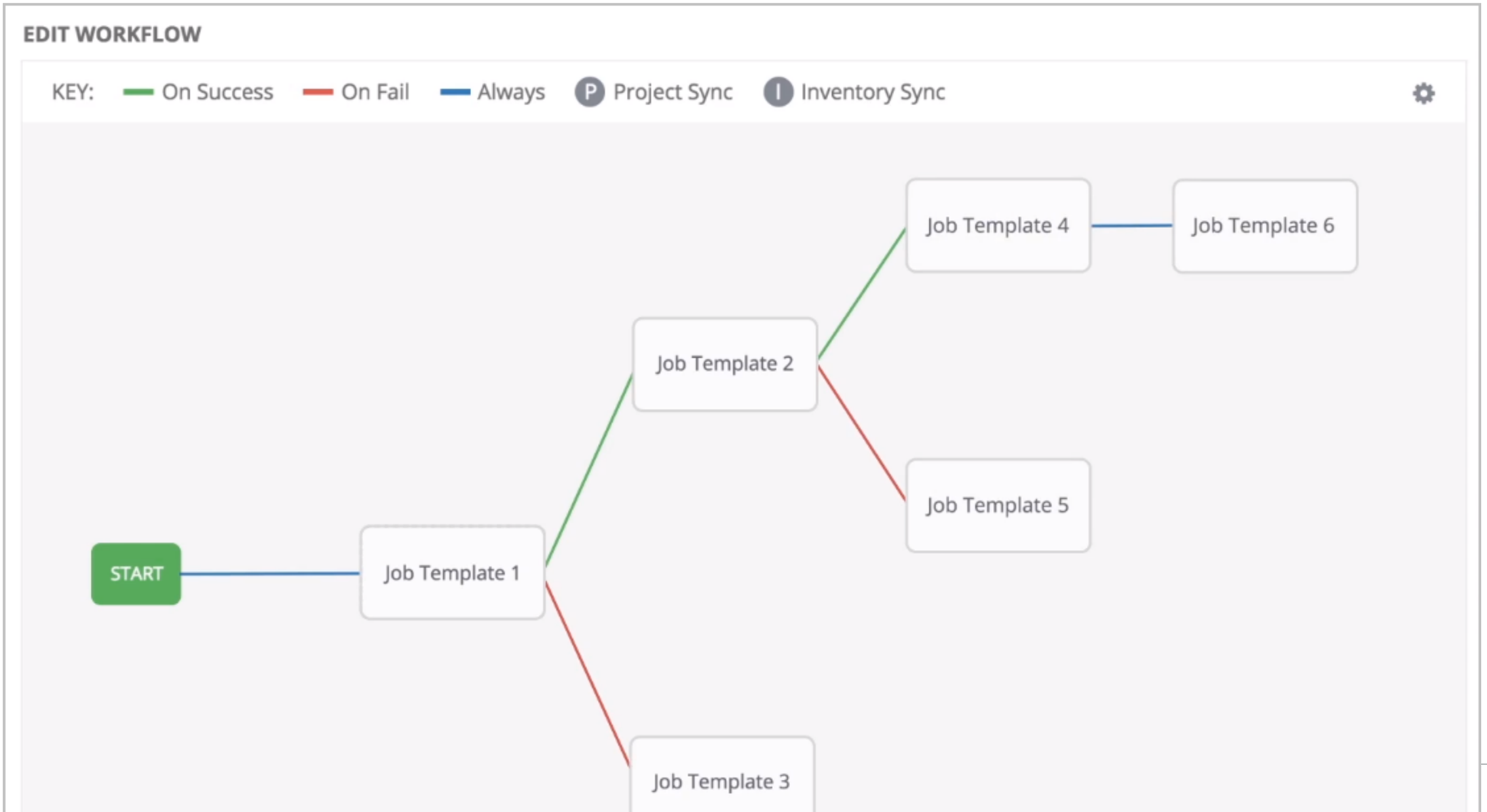
Deploy application

PLAYS 1 TASKS 4 HOSTS 2 ELAPSED 00:00:17

SEARCH KEY

```
PLAY [all] ********************************************************************* 18:52:29

TASK [Check required permissions] ********************************************** 18:52:29
ok: [18.218.56.237]
ok: [18.224.239.56]

TASK [Fix permissions if needed] *********************************************** 18:52:30
skipping: [18.218.56.237]
skipping: [18.224.239.56]

TASK [Deploy updates] ********************************************************** 18:52:30
changed: [18.224.239.56]
changed: [18.218.56.237]

TASK [Restart service] ********************************************************* 18:52:33
changed: [18.224.239.56]
changed: [18.218.56.237]

PLAY RECAP ********************************************************************* 18:52:35
18.218.56.237              : ok=3    changed=2    unreachable=0    failed=0
18.224.239.56              : ok=3    changed=2    unreachable=0    failed=0
```

MULTI-PLAYBOOK WORKFLOWS



WHO RAN WHAT JOB WHEN

TOWER

admin

ACTIVITY STREAM

ACTIVITY STREAM | **ALL ACTIVITY**

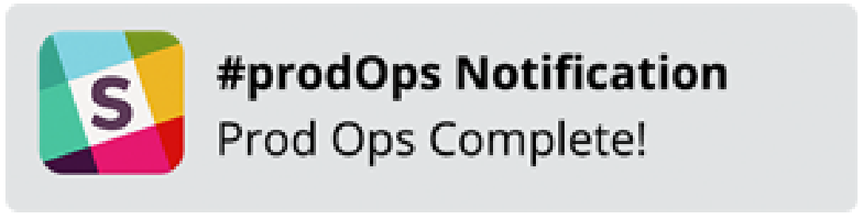
SEARCH

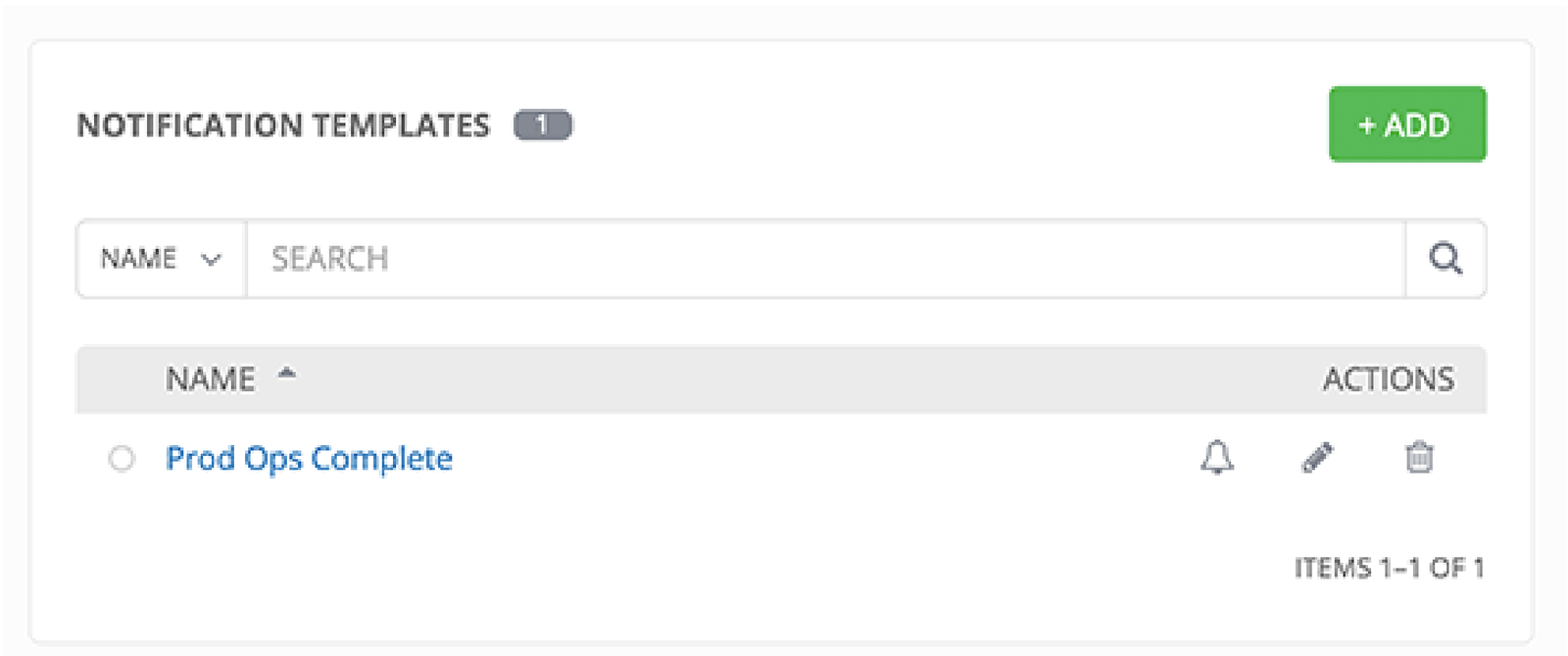
KEY

All Activity

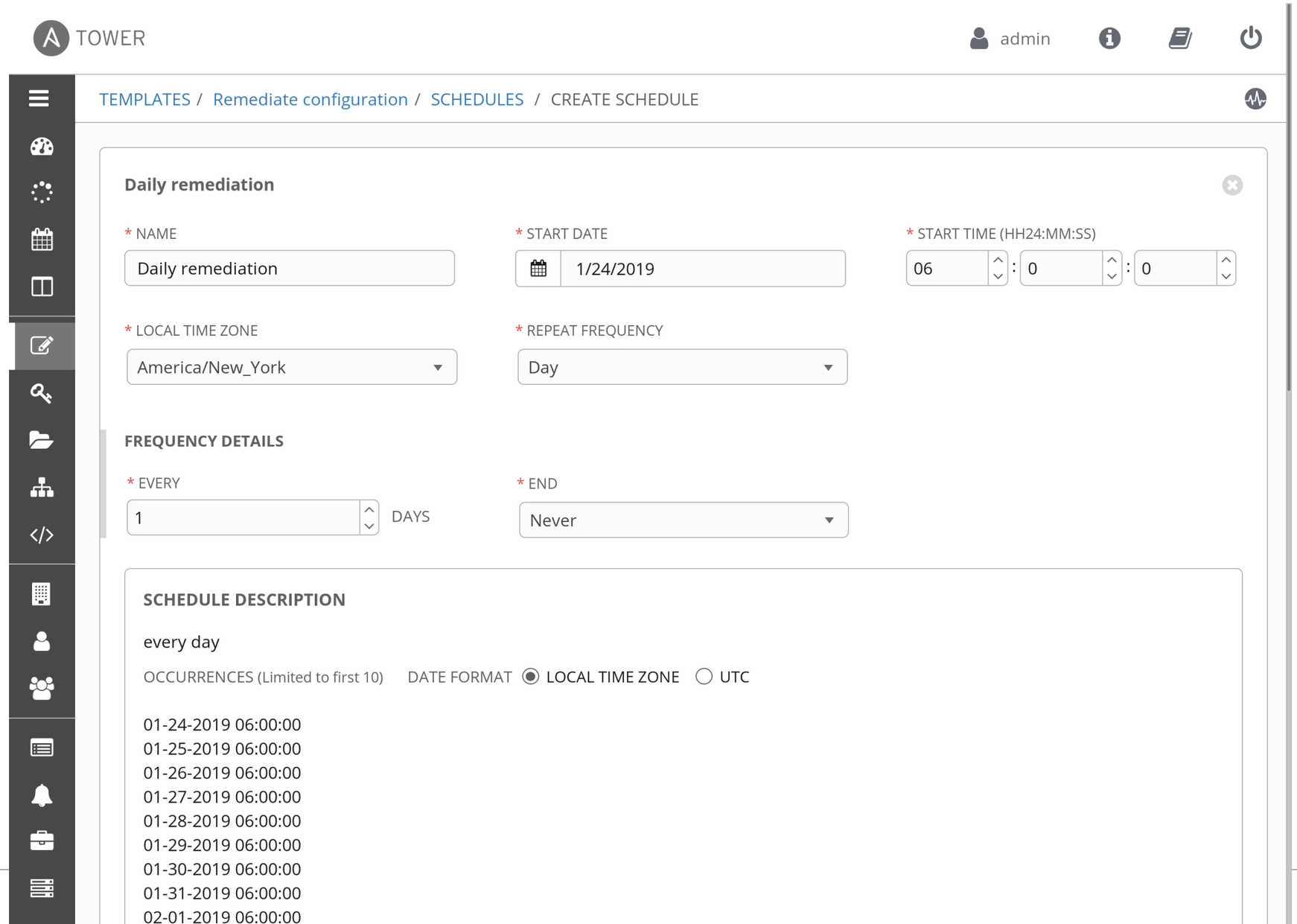
TIME	INITIATED BY	EVENT	ACTIONS
1/14/2019 11:27:20 AM	system	created job Run tests	
1/14/2019 11:27:19 AM	system	created workflow_job Deploy database	
1/14/2019 11:27:18 AM	system	created workflow_job Deploy application	

INTEGRATED NOTIFICATIONS

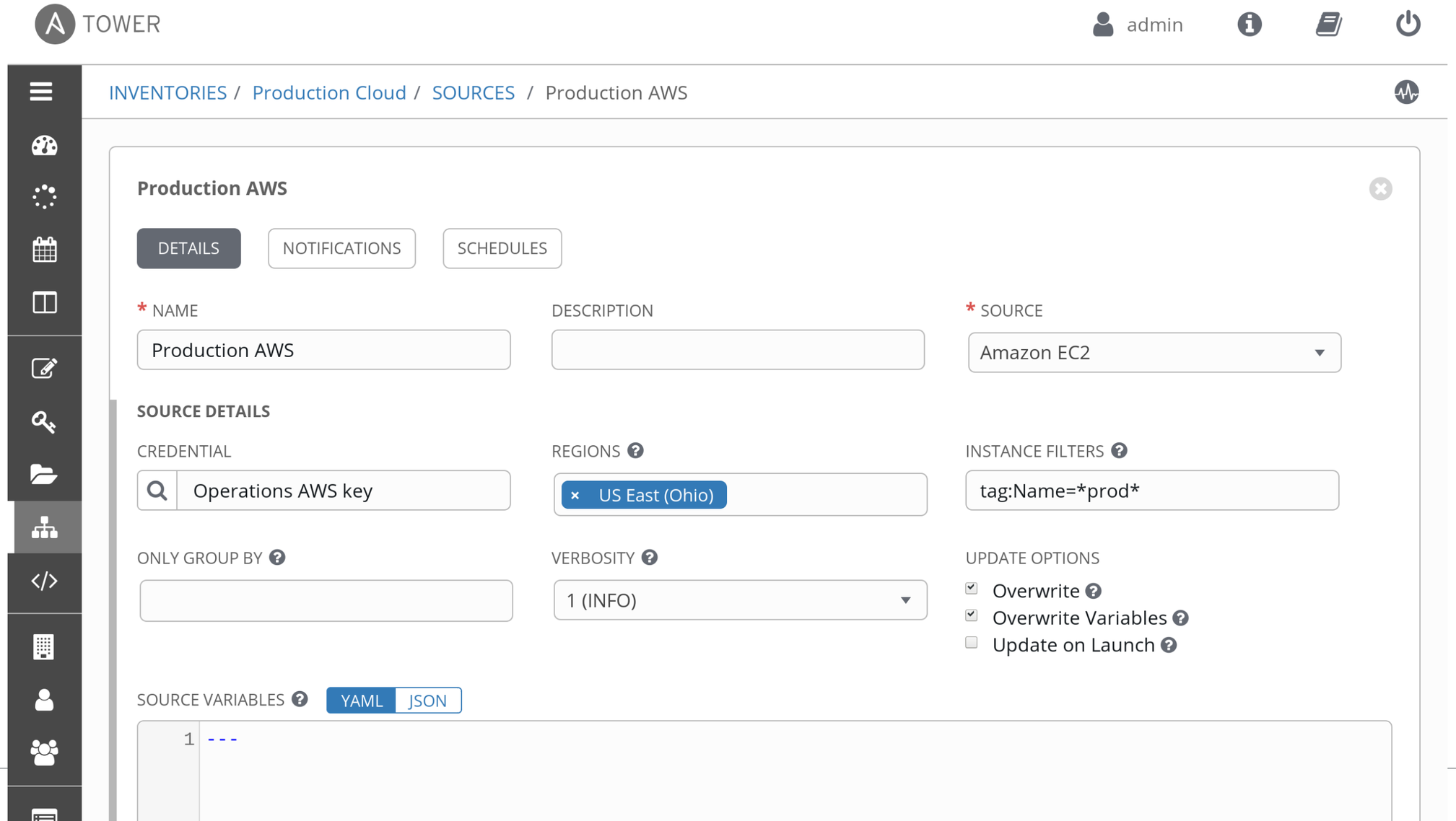




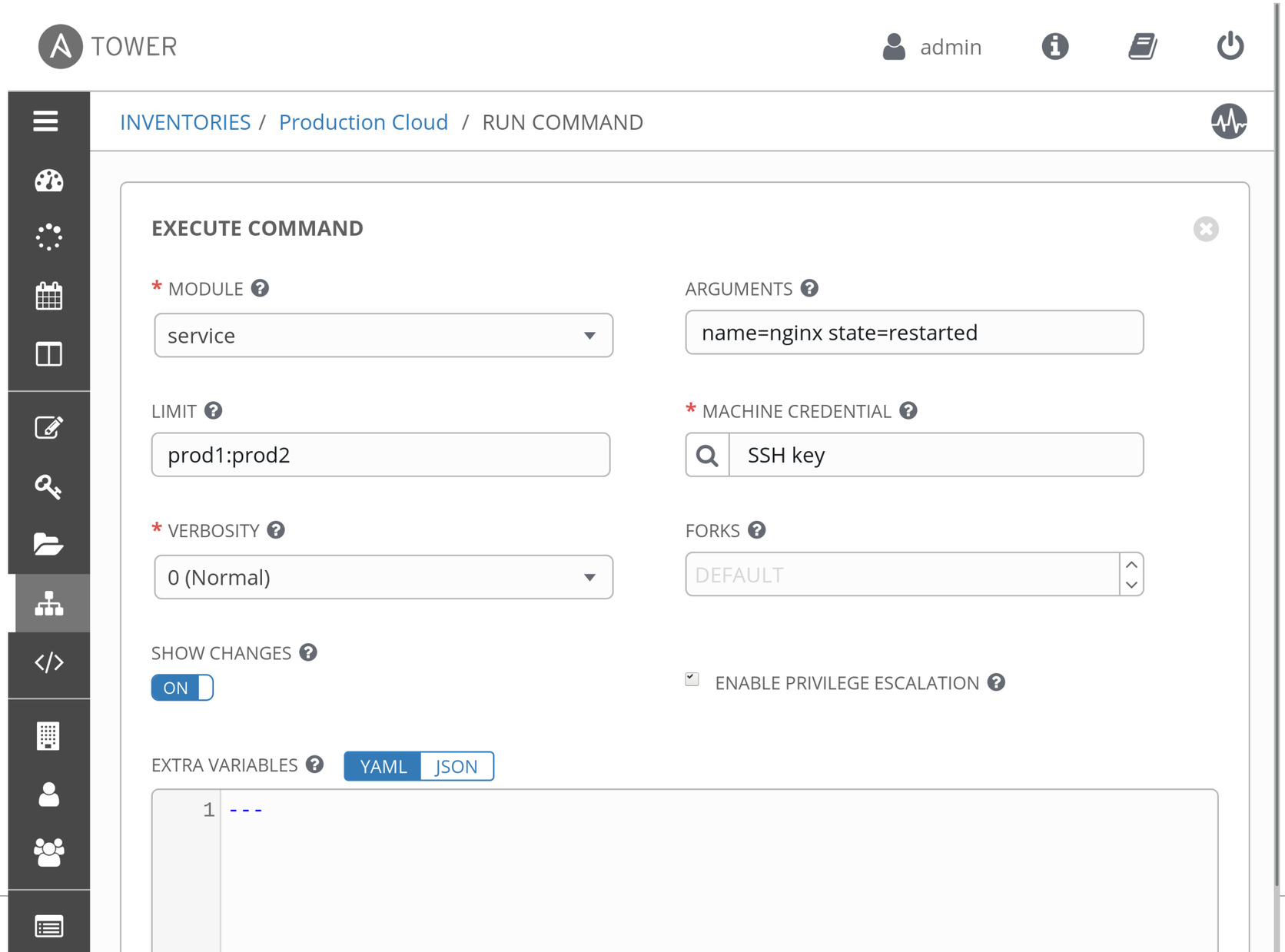
SCHEDULE ANSIBLE JOBS



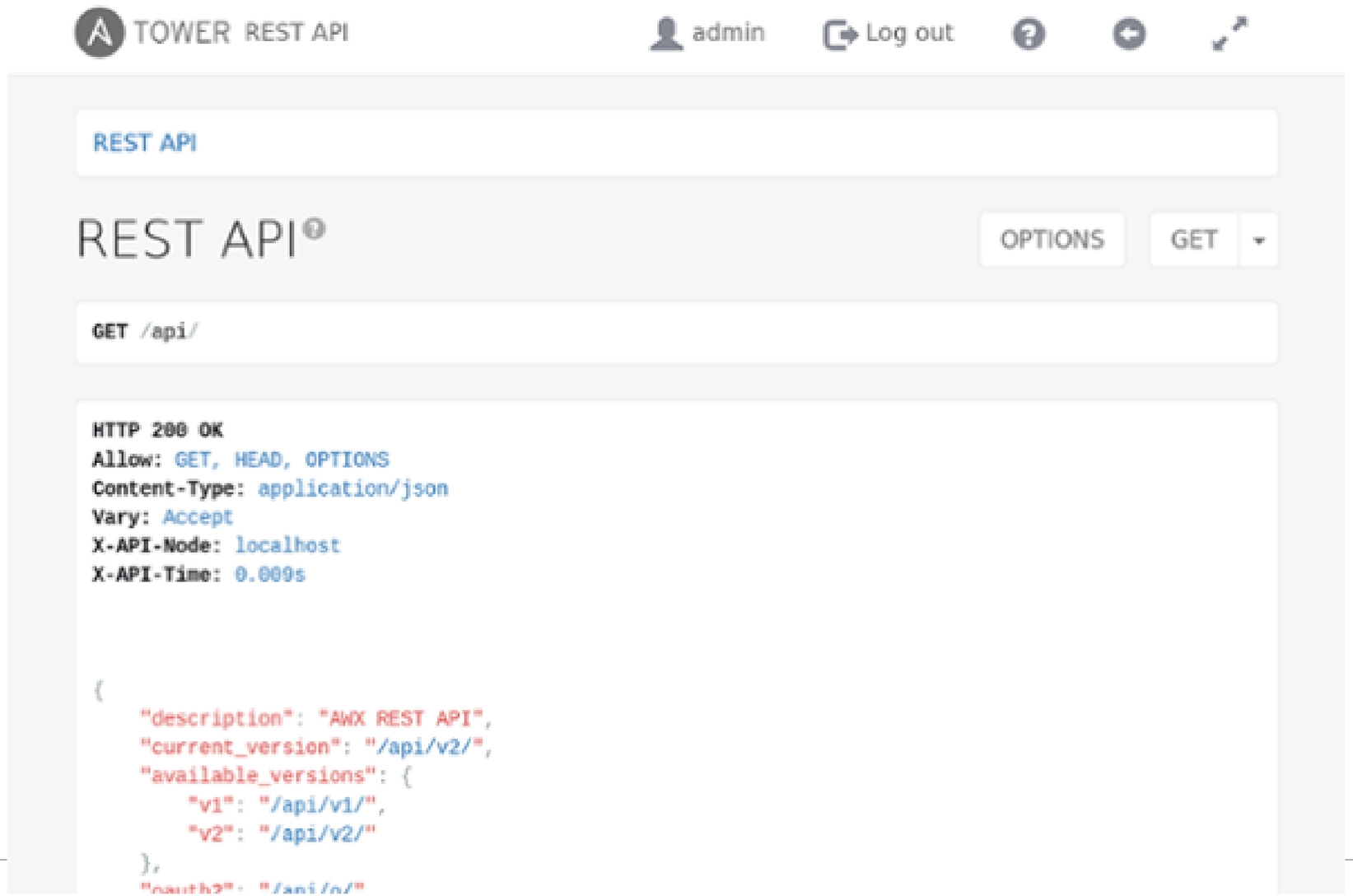
MANAGE AND TRACK YOUR ENTIRE INVENTORY



REMOTE COMMAND EXECUTION



COMPREHENSIVE REST API AND TOWER CLI TOOL



OneCloud Consulting

The Cloud Services Division of ePlus

Variables, Facts & Inclusions

Module 7

Objectives

+ Manage variables in Ansible
+ Manage Facts in Playbooks
+ Include variables and tasks from external files into a playbook
+ Discuss loops and how they are used.

Ansible Variables

Ansible Variables

- Ansible supports **variables**
- Variables can be used to **store values** that can be reused throughout files in an entire Ansible project.
- Simplify **creation** and **maintenance** of a project
- Reduce the incidence of **errors**
- Provide a convenient way to manage **dynamic values** for a given environment

Variables

Ansible can work with metadata from various sources and manage their context in the form of variables

+ Command line parameters
+ Plays and tasks
+ Files
+ Inventory
+ Discovered facts
+ Roles

What variables might include

- Users to create
- Packages to install
- Services to restart
- Files to remove
- Archives to retrieve from the Internet

Naming Variables

- Names which consist of a **string**
- Start with a **letter** and can only contain **letters**, **numbers**, and **underscores**

Invalid variable names	Valid variable names
`web server`	`web_server`
`remote.file`	`remote_file`
`1st file`	`file_1 or file1`
`remoteserver$1`	`remote_server_1 or remote_server1`

Defining Variables and Scopes

- Variables can be defined in a bewildering variety of places in an Ansible project.
- Simplified in to three **basic scope levels**:
  - ***Global scope*:**
    - Variables set from the command line or Ansible configuration
  - ***Play scope*:**
    - Variables set in the play and related structures
  - ***Host scope*:**
    - Variables set on host groups and individual hosts by the inventory, fact gathering, or registered tasks

Variables in Playbooks

How do you define variables in playbooks?

- Playbook variables can be defined in **multiple ways :**

vars

```
- hosts: all
  vars:
     user: joe
     home: /home/joe
```

vars_files

```
- hosts: all
   vars_files:
     - vars/users.yml
```

Variables in Playbooks ..

How do you Use Variables in Playbooks?

- Variables are referenced by placing the variable name in double curly braces.
- Ansible substitutes the variable with its value when the task is executed.
- When a variable is used as the first element to start a value, quotes are mandatory.

```
- hosts: all
  vars:
    user: joe
  tasks:
    - name: Creates the user {{ user }}
      user:
        name: "{{ user }}"
```

Host Variables and Group Variables

Inventory variables for managed hosts fall into **two broad categories**:

host variables

- apply to a specific host

group variables

- apply to all hosts in a host group or in a group of host groups.

Host variables take **precedence** over group variables, defined by a playbook take precedence over both.

Host and Group Variables example

Host Variable

```
[servers]
demo.example.com ansible_user=joe
```

Group Variables

```
[servers]
demo1.example.com
demo2.example.com

[servers:vars]
ansible_user=joe
```

Secure Variables

- Encrypt variables, tasks, handlers, etc. using Ansible Vault:

```
# ansible-vault create vars/secure.yml
# ansible-vault edit vars/secure.yml

# ansible-playbook --vault-password-file
/etc/ansible/pw /etc/ansible/monitoring.yml
```

Overriding Variables from the Command Line

- Inventory variables are overridden by variables set in a playbook
- Variables may be overridden through arguments passed to the **ansible** or **ansible-playbook** commands
- Defined value for a variable needs to be overridden for a single host for a one-off run of a playbook.


```
ansible-playbook main.yml --limit=demo2.example.com
-e "package=apache"
```

Variables and Arrays

How to declare an array variable in Ansible?

+ List of **variables values** assigned into an **array**

```
- hosts: all
  vars:
    hello:
      - World
      - Asia
      - South America
      - North America
      - Artic
      - Antartic
      - Oceania
      - Europe
      - Africa
tasks:
- name: Ansible List variable Example
  debug:
    msg: "{{ hello[2] }}"
```

1

5

Variables and Arrays

Also define the variables using **with_items** structure to loop through all the values :

```
- hosts: all
  vars:
    hello: [Asia, Americas, Artic, Antartic,
Oceania, Europe, Africa]
  tasks:
  - name: Ansible array variables example
    debug:
      msg: "{{ item }}"
    with_items:
      - "{{ hello }}"
```

Registered Variables

```
---
- name: Installs a package and prints
the result
  hosts: all

  tasks:
    - name: Install the package
      yum:
        name: httpd                    1
        state: installed               7
      register: install_result
      - debug: var=install_result
```

- Ansible registers are used to capture the output of a task (debugging purposes) to a variable.
- Can use these register in different scenarios
  - Conditional statement
  - Logging
- Variables will contain the value returned by the task
- Modules like shell and yum have specific return values
- Each registered variable will be valid on the remote host where the task was run for the rest of the playbook execution.

Magic Variables

- Some variables are **not facts** or configured through the **setup** module
- These ***magic variables*** can also be useful to get information specific to a particular managed host.
- Magic varaibles Features:
  - **hostvars**
  - **group_names**
  - **groups**
  - **inventory_hostname**

```
ansible localhost -m debug -a 'var=hostvars["localhost"]'
```

Ansible 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 like regular variables
  - Plays
  - Conditionals
  - Loops,
- Any other statement that depends on a value collected from a managed host.

2

0

What a managed host might include

- Host name
- Kernel version
- Network interfaces
- IP addresses
- Operating system version [2]
- Various environment variables [1]
- Number of CPUs
- Available or free memory
- Available disk space

Facts

Facts are a convenient way to retrieve the state of a managed host and to determine what action to take based on that state.

- A server can be restarted by a conditional task which is run based on a fact containing the managed host's current kernel version.

- The MySQL configuration file can be customized depending on the available memory reported by a fact.

- The IPv4 address used in a configuration file can be set based on the value of a fact.

Facts variables

Facts	Variables
Short hostname	`ansible_hostname`
Fully-qualified domain name	`ansible_fqdn`
Main IPv4 address (based on routing) [2]	`ansible_default_ipv4.address`
A list of the names of all network interfaces [3]	`ansible_interfaces`
Main disk first partition size (based on disk name, such as **vda**, **vdb**, and so on.)	`ansible_devices.vda.partitions.vda1.size`
A list of DNS servers	`ansible_dns.nameservers`
Version of the currently running kernel	`ansible_kernel`

Facts in playbook

+ Ansible dynamically substitutes the variable name with the corresponding value:

```
---
- hosts: all

  tasks:
    - name: Prints various Ansible facts
      debug:
        msg: >
          The default IPv4 address of {{ ansible_fqdn }} is {{ ansible_default_ipv4.address }}
```

Turning Off Fact Gathering

- Some case, don't want to gather facts for your play.
- There are a **couple of reasons** why this might be the case.
  - It might be that you are not using any facts and want to speed up the play
  - reduce load caused by the play on the managed hosts.
  - It might be that the managed hosts can't run the **setup** module for some reason, or need to install some perquisite software before gathering facts.

Turning Off Fact Gathering ..

+ To disable fact gathering for a play, set the **gather_facts** key to **no**:

```
---
- name: This play gathers no facts automatically
  hosts: large_farm
  gather_facts: no
```


+ Even if **gather_facts: no** is set for a play, you can manually gather facts at any time by running a task that uses the **setup** module:

```
tasks:
    - name: Manually gather facts
      setup:
```

Fact Filters

- Ansible facts contain **extensive** information about the system.
- Administrators can use Ansible ***filters*** in order to **limit** the results returned when **gathering facts** from a managed node.
- Filters can be used to:
  - Only retrieve information about network cards.
  - Only retrieve information about disks.
  - Only retrieve information about users.

Fact Filters expression

- To use filters, the expression needs to be passed as an option, using **-a 'filter=*EXPRESSION*'**.
- For example, to only return information about **eth0**, a filter can be applied on the **ansible_eth0** element:

```
# ansible demo1.example.com -m setup -a 'filter=ansible_eth0'
```

Custom Facts

- Administrators can create ***custom facts*** which are stored locally on each managed host.
- Facts are integrated into the list of standard facts gathered by **setup** when it runs on the managed host.
- Allows the managed host to provide **arbitrary variables** to Ansible which can be used to adjust the behavior of **plays**.

- **Custom facts** can be defined in a **static file**, formatted as an **INI** file or using **JSON.**
- They can also be **executable scripts** which generate **JSON** output, just like a **dynamic inventory script**.

Custom Facts (continued)

- By default, **setup** loads custom facts from files and scripts in each managed host's **/etc/ansible/facts.d** directory.
- Script file must end with **.fact**
- Dynamic custom fact scripts must output **JSON-formatted** facts and must be executable.

```
[packages]
web_package = httpd
db_package = mariadb-server
[users]
user1 = joe
user2 = jane
```

```
{"packages": {
"web_package": "httpd",
"db_package": "mariadb-
server"},
"users": {
"user1": "joe",
"user2": "jane"
}}
```

Ansible Inclusions

Inclusions

- Separate files to divide tasks and lists of variables into smaller pieces for easier management.
- Multiple ways to include task files and variables in a playbook
  - Tasks can be included in a playbook from an external file by using the **include** directive.

```
tasks:
   - name: Include tasks to install the database server
     include: tasks/db_server.yml
```

Inclusions (continued)

+ The **include_vars** module can include variables defined in either JSON or YAML files, overriding host variables and playbook variables already defined.

```
tasks:
   - name: Include the variables from a YAML or JSON file
     include_vars: vars/variables.yml
```

+ Using multiple, external files for tasks and variables is to build the main playbook in a modular way.

Including Tasks

- Manage a set of tasks as a file separate from the playbook
- In **Ansible 2.4**, the **include** module is **deprecated.**
- Ships with two replacement modules**, import_tasks** and **include_tasks**
- **include_tasks:** Includes a file with a list of tasks to be executed in the current playbook.
- **import_tasks:** Imports a list of tasks to be added to the current playbook for subsequent execution.

Including Variables

- Variables can be externally defined and included in playbooks
- Different ways set of variables include
  - Inventory variables defined in the **inventory** file or in **external** files in **host_vars** and **group_vars** directories
  - **Facts** and **registered** variables
  - Playbook variables defined in the playbook file with **vars** or in an external file through **vars_files**

include_vars

- The ***include_vars*** module is a set variables in a playbook from an **external** file.
- Useful, when combining task files that set values for their variables which you want to override
- Included variable files may be defined using either **JSON** or **YAML**, YAML being the preferred syntax.
- The **include_vars** module takes the path of the variable file to import as an argument.

include_vars (continued)

+ Define variables and include them in a playbook with **include_vars**.

```
---
- name: Install web application packages
  hosts: all
  tasks:
    - name: Includes the tasks file and defines the variables
      include_vars: variables.yml
    - name: Debugs the variables imported
      debug:
        msg: "{{ packages['web_package'] }} and {{ packages.db_package }} have been imported"
```

Summary

- Ansible *variables* allow administrators to reuse values across files in an entire Ansible project
- Variables have names which consist of a string that must start with a letter and can only contain letters, numbers, and underscores
- Variables can be defined for hosts and host groups in the inventory, for playbooks, by facts and external files, and from the command line

- It is better to store inventory variables in files in the **host_vars** and **group_vars** directory relative to the inventory than in the inventory file itself
- Ansible *facts* are variables that are automatically discovered by Ansible from a managed host

Summary (continued)

- In a playbook, when a variable is used at the start of a value, quotes are mandatory
- The **register** keyword can be used to capture the output of a command in a variable.
- Both **include** and **include_vars** modules can be used to include tasks or variable files in YAML or JSON format in playbooks.

Lab

Managing Variables and Inclusions

Loops & Task Control

Module 8

Objectives

- Implement loops in playbooks
- Construct conditionals in playbooks
- Combine loops and conditionals
- Implement handlers in playbooks
- Set tags to control which parts of a playbook should be run
- Resolve errors in a playbook

Task Iteration with Loops

- Ansible supports several different ways to iterate a task over a set of items using a loop.
- Loops can repeat a task using each item in a list, the contents of each of the files in a list, a generated sequence of numbers, or using more complicated structures.
- Using loops saves administrators from the need to write multiple tasks that use the same module.
- For example, instead of writing five tasks to ensure five users exist, one task can be written that iterates over a list of five users to ensure they all exist.

Loops

- A simple loop iterates a task over a list of items.
- The **loop** key 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 current value being used for this iteration.

```
- name: Add Multiple Vlans
    nxos_vlan:
      vlan_id: "{{ item }}"
      state: present
    loop:
      - 50
      - 60
      - 70
```

Nested Loops

```
---


- name: Loop Example
  hosts: nxos
  tasks:
    - name: Add Multiple Vlans
      nxos_vlan:
        vlan_id: "{{ item[0] }}"
        name: "{{ item[1] }}"
        state: present
      loop:
        - ['101', 'prod']
        - ['102', 'dev']
```

+ The **loop** key can also be used for nested loops, loops run inside of loops. It takes a list of two or more lists.
+ For example, given a list of two lists, the task will iterate each item in the first list in combination with each item in the second list.
+ To illustrate this better, look at the following snippet from a play.
+ The **nxos_vlan** module is used to create vlan's in the first list with using the second list for the vlan name. This is a nested loop example.

Running Tasks Conditionally

- Ansible can use *conditionals* to execute tasks or plays when certain conditions are met.
- For example, a conditional can be used to determine the available memory on a managed host before Ansible installs or configures a service.
- Conditionals allow administrators to differentiate between managed hosts and assign them functional roles based on the conditions that they meet.
- Playbook variables, registered variables, and Ansible facts can all be tested with conditionals.
- Operators such as string comparison, mathematical operators, and Booleans are available.

Examples of Conditionals

- A hard limit can be defined in a variable (for example, **min_memory**) and compared against the available memory on a managed host.
- The output of a command can be captured and evaluated by Ansible to determine whether or not a task completed before taking further action.
- Ansible facts can be used to determine the managed host network configuration and decide which template file to send (for example, network bonding or trunking).
- Verify a certain NXOS version before applying a command or patch.
- Use conditions to check and ensure certain ports are up before trying to configure them.

Using lookup vs query with loop

- In Ansible 2.5 a new jinja2 function was introduced named ‘query’, that offers several benefits over lookup when using the new loop keyword.
- Query provides a more simple interface and a more predictable output from lookup plugins, ensuring better compatibility with loop.

```
loop: "{{ query('inventory_hostnames', 'all') }}"

loop: "{{ lookup('inventory_hostnames', 'all', wantlist=True) }}"
```

Ansible **when** Statement

- 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 which can be tested is whether a Boolean variable is true or false.
- The **when** statement in the following example causes the task to run only if **run_my_task** is true:

```
---
- hosts: all
  vars:
     run_my_task: true
tasks:
    - name: httpd package is installed
      yum:
         name: httpd
      when: run_my_task
```

Testing Multiple Conditions

- One **when** statement can be used to evaluate multiple values.
- Conditionals can be combined with the **and** and **or** keywords or grouped with parentheses.
- Expressing multiple conditions in different ways:
  - `ansible_kernel == 3.10.0-327.el7.x86_64 and inventory_hostname in groups['staging']`
  - `ansible_distribution == "RedHat" or ansible_distribution == "Fedora"`
  -
```
(ansible_distribution == "RedHat" and ansible_distribution_major_version == 7) or
(ansible_distribution == "Fedora" and ansible_distribution_major_version == 23)
```

Combining Loops and Conditional Tasks

- Loops and conditionals can be combined.
  - Example, the *ntp configuration* will be applied only if ntp is enabled, etc.

- The loop iterates over each dictionary in the list, and the conditional statement is not met unless a dictionary is found representing a mounted file system where both conditions are true.

Example

- Combining conditionals and registered variables
- Annotated playbook will restart the **httpd** service only if the **postfix** service is running.

```
- hosts: all
  tasks:
    - name: Postfix server status
      command: /usr/bin/systemctl is-active postfix
      ignore_errors: yes
      register: result

  - name: Restart Apache HTTPD if Postfix running
    service:
      name: httpd
      state: restarted
    when: result.rc == 0
```

1 Is Postfix running or not?

2 If it is not running and the command "fails", do not stop processing

3 Saves information on the module's result in a variable named **result**

4 Evaluates the output of the Postfix task. If the exit code of the **systemctl** command is 0, then Postfix is active and this task will restart the **httpd** service.

Ansible Handlers

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

Ansible Handlers ..

- *Handlers* are tasks that respond to a notification triggered by other tasks.
- Each handler has a globally-unique name, and is triggered at the end of a block of tasks in a playbook.
- If no task notifies the handler by name, it will not run.
- If one or more tasks notify the handler, it will run exactly once after all other tasks in the play have completed.
- Because handlers are tasks, administrators can use the same modules in handlers that they would for any other task.
- Handlers are used to reboot hosts and restart services.

Ansible Handlers Example

```
tasks:
  - name: copy demo.example.conf configuration template (1)
    copy:
      src: /var/lib/templates/demo.example.conf.template
      dest: /etc/httpd/conf.d/demo.example.conf
    notify: (2)
      - restart_apache (3)

handlers: (4)
  - name: restart_apache (5)
    service: (6)
      name: httpd
      state: restarted
```

(1) → The task that notifies the handler

(2) → The **notify** statement indicates the task needs to trigger a handler

(3) → The name of the handler to run.

(4) → The statement starts the handlers section.

(5) → The name of the handler invoked by tasks.

(6) → The module to use for the handler.

Using Handlers

- Handlers are always run in the order in which the **handlers** section is written in the play, not in the order in which they are listed by the **notify** statement on a particular task.
- Handlers run after all other tasks in the play complete. A handler called by a task in the **tasks:** part of the playbook will not run until *all* of the tasks under **tasks:** have been processed.
- Handler names live in a global namespace. If two handlers are incorrectly given the same name, only one will run.
- Handlers defined inside an **include** can not be notified.
- Even if more than one task notifies a handler, the handler will only run once. If no tasks notify it, a handler will not run.
- If a task that includes a **notify** does not execute (for example, a package is already installed), the handler will not be notified. The handler will be skipped unless another task notifies it.
- Ansible notifies handlers only if the task acquires the **CHANGED** status.

Tagging Ansible Resources

- *Tags* can be applied to specific resources as a text label in order to allow this.
- Tagging a resource only requires that the **tags** keyword be used, followed by a list of tags to apply.
- When plays are tagged, the **--tags** option can be used with **ansible-playbook** to filter the playbook to only execute specific tagged plays.

Tag Example:

+ In playbooks, each task can be tagged, using the **tags** keyword:

```
tasks:
  - name: Package {{ item }} is installed
    yum:
      name: "{{ item }}"
      state: installed
    with_items:
      - postfix
      - mariadb-server
    tags:
      - packages
```


+ When a task file is included in a playbook, the task can be tagged, allowing administrators to set a global tag for the **include** statement:

```
- include: common.yml
  tags:
    - webproxy
    - webserver
```

Managing Tagged Resources

- When a tag has been applied to a resource, the **ansible-playbook** command can be used with the **--tags** or **--skip-tags** argument to either execute the tagged resources, or prevent the tagged resources from being included in the play.

```
---
- name: Example play using tagging
  hosts:
    - servera.lab.example.com
    - serverb.lab.example.com

  tasks:
    - name: httpd is installed
      yum:
        name: httpd
        state: latest
      tags: webserver

    - name: postfix is installed
      yum:
        name: postfix
        state: latest
```

Managing Tagged Resources

- To only run the first task, the **--tags** argument can be used:

```
# ansible-playbook main.yml --tags 'webserver'
```

- **--tags** option was specified, the playbook only ran the task tagged with the **webserver** tag.
- To skip tasks with a specific tag and only run the tasks without that tag, the **--skip-tags** option can be used:


```
# ansible-playbook main.yml --skip-tags 'webserver'
```

Special Tags

- Ansible has a special tag that can be assigned in a playbook: **always**.
- This tag causes the task to always be executed even if the **--skip-tags** option is used, unless explicitly skipped with **--skip-tags always**.
- **Three** special tags using CLI with **--tags** option :
  - The **tagged** keyword is used to run any tagged resource.
  - The **untagged** keyword does the opposite of the **tagged** keyword by excluding all tagged resources from the play
  - The **all** keyword allows administrators to include all tasks in the play. This is the defaultbehavior of the command line.

Handling Errors

- Ansible evaluates the return code of each task to determine whether the task succeeded or failed.
- When a task fails Ansible immediately aborts the rest of the play on that host, skipping all subsequent tasks.
- Ansible features that can be used to manage task errors:
  - Ignoring Task Failure
  - Forcing Execution of Handlers after Task Failure
  - Specifying Task Failure Conditions
  - Specifying When a Task Reports "Changed" Results
  - Ansible Blocks and Error Handling

2

Summary

- *Loops* can be used to iterate over a set of values.
- They can be a list of items in sequence or random order, files, an auto generated sequence of numbers, or other things
- *Conditionals* can be used to execute tasks or plays only when certain conditions have been met
- Conditions can be tested with various operators, including string comparisons, mathematical operators, and Boolean values
- *Handlers* are special tasks that execute at the end of the play if notified by other tasks
- *Tags* are used to mark certain tasks to be skipped or executed based on what tags a task has

Summary

- Tasks can be configured to handle error conditions by ignoring task failure, forcing handlers to be called even if the task failed, mark a task as failed when it succeeded, or override the behavior that causes a task to be marked as changed
- *Blocks* can be used to group tasks as a unit and execute other tasks depending on whether or not all the tasks in the block succeed

Lab 8

Loops

Ansible Templating

Module 12

Objectives

- Why templates?
- Describe Jinja2 templates
- Describe variables, control structures, and comment use in Jinja2 templates
- Build a template file
- Use the template file in a playbook

Why Templates?

- Templates are simple text files that we can use in Ansible
- A template is a file which contains all your configuration parameters, but the dynamic values are given as variables.
- Playbook execution checks with the conditions, like
  - Cluster
  - Variables
  - Conditional statements
  - Loops
  - Filters

Jinja2 Templates

- Ansible uses the **Jinja2** templating system to modify files before they are distributed to managed hosts.
- Templates can be useful when systems need to have slightly modified versions of the same file.
- Ansible also uses **Jinja2** to reference **variables** in playbooks.
- Ansible **allows Jinja2 loops** and **conditionals** to be used in templates, but they are **not allowed** in playbooks.
- Ansible playbooks are completely **machine-parseable YAML**.

Jinja2 Templates

- Jinja2 is a library for Python that is designed to be flexible, fast and secure.
- Text-based template languages
- It’s both designer and developer friendly by sticking to Python’s principles
- Jinja2 works with Python >= 2.6.x
- Jinja 2.6v support for Python 3.2

Jinja Delimiters

- Variables or logic expressions are placed between tags, or delimiters.
- **{% ... %}** for Statements
- **{{ ... }}** for Expressions to print to the template output
- **{# ... #}** for Comments not included in the template output
- **# ... ##** for Line Statements

Jinja2 Execution Structure

```
{# /etc/hosts line #}
{% for user in users %}
  {{ user }}
{% endfor %}
```

Jinja Templating Features

- Sandboxed execution
- Powerful automatic HTML escaping system for XSS prevention
- Template inheritance
- Compiles down to the optimal python code just in time
- Optional ahead-of-time template compilation
- Easy to debug
- Line numbers of exceptions directly point to the correct line in the template.
- Configurable syntax

template module

- **src:**
  - The value associated with the **src** key specifies the **source Jinja2** template
- **dest:**
  - The value associated to the **dest** key specifies the file to be created on the **destination hosts**.

```
tasks:
  - name: template render
    template:
      src: /tmp/j2-template.j2
      dest: /tmp/dest-config-file.txt
```

Variable Filters

```
{{ output | to_json }}
{{ output | to_yaml }}

{{ output | to_nice_json }}
{{ output | to_nice_yaml }}

{{ output | from_json }}
{{ output | from_yaml }}
```

- Jinja2 provides **filters** which change the **output format** for template expressions (for example, to JSON).
- There are filters available for languages such as **YAML** or **JSON**.
- **to_json** filter formats the expression output using JSON,
- **to_yaml** filter uses YAML as the formatting syntax.
- **to_nice_json filters** format the expression output in JSON
- **to_nice_yaml** filters format the expression output in YAML
- **from_json** filters expect a string in JSON
- **from_yaml** filters expect a string in YAML

Building a Jinja2 Template

- A Jinja2 template is composed of multiple elements: **data**, **variables** and **expressions**.
- Those **variables** and **expressions** are replaced with their values when the Jinja2 template is **rendered**.
- The variables used in the template can be specified in the **vars** section of the playbook.
- It is possible to use the **managed hosts' facts** as variables on a template

```
ansible system_hostname -i inventory_file -m setup
```

Using Jinja2 Templates in Playbooks

- Jinja2 templates are a powerful tool
- 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,
- Supports the transfer of a local file in the control node to the managed hosts.
- Use **template** module has **two** variables:
  - **src**
  - **dest**

Summary

- Ansible uses the Jinja2 templating system to render files before they are distributed to managed hosts.
- YAML allows the use of Jinja2 based variables, with or without filters, inside the definition of a playbook.
- A Jinja2 template is usually composed of two elements: variables and expressions.
- Those variables and expressions are replaced with their values when the Jinja2 template is rendered.
- Filters in Jinja2 are a way of transforming template expressions from one kind of data into another.

Lab 9

Jinja 2 Templating

OneCloud
Consulting
The Cloud Services Division of ePlus

Ansible Roles

Module 10

Objectives

- Describe the structure and behavior of a role
- Define role dependencies
- Create an Ansible role

Ansible Roles

Why Roles?

- Roles provide Ansible with a way to load **tasks**, **handlers**, and **variables** from **external files**.
- **Static files** and **templates** can also be associated and referenced by a **role**.
- The files that define a role have specific **names** and are organized in a rigid **directory** structure, which will be discussed later.
- **Roles** can be written so they are general purpose and can be reused.

Benefits of Ansible roles

- Roles group content, in particular tasks & templates, allowing easy sharing of code with others
- Roles can be written that define the essential elements of a system type:
  - switches
  - routers
  - Git repository
  - Other purpose
- Roles make larger projects more manageable
- Roles can be developed in parallel by different administrators

Ansible role structure

- An Ansible **role's** functionality is defined by its **directory structure**.
- The **top-level directory** defines the name of the role itself.
- Subdirectories contain **YAML** files, named **main.yml**.
- The **files** and **templates** subdirectories can contain objects referenced by the YAML files.

```
# tree user.example
```

tree – Role Structure

```
user.example/
├── defaults
│   └── main.yml
├── files
├── handlers
│   └── main.yml
├── meta
│   └── main.yml
├── README.md
├── tasks
│   └── main.yml
├── templates
├── tests
│   ├── inventory
│   └── test.yml
└── vars
    └── main.yml
```

Ansible Role Subdirectories

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.
`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.

Ansible roles in a playbook

+ Using roles in a playbook are straightforward

```
---
- hosts: remote.example.com
  roles:
  - role1
  - role2
```

+ For each role specified, the role **tasks**, role **handlers**, role **variables**, and role **dependencies** will be included in the playbook

+ Any **copy**, **script**, **template**, or **include** tasks in the role can reference the relevant **files**, **templates**, or **tasks** without absolute or relative path names.

Defining role dependencies

- Role dependencies allow a role to include other roles as dependencies in a playbook.
  - For example, a role that defines a documentation server may depend upon another role that installs and configures a web server.
- Dependencies are defined in the **meta/main.yml** file in the role directory hierarchy.

1
0

```
---
  dependencies:
    - { role: apache, port: 8080 }
    - { role: postgres, dbname: serverlist, admin_user: felix }
```

How to create roles?

- Creating and using a role is a **three** step process:
  - Create the role directory structure
  - Define the role content
  - Use the role in a playbook

1

Creating the role directory structure

- Ansible looks for roles in a subdirectory called **roles** in the project directory.
- Roles can be the directories referenced by the **roles_path** variable in Ansible config files.
- This variable contains a **colon-separated list** of directories to search.
- Each role has its own directory with specially named subdirectories

```
# tree roles/

roles/
└── motd
    ├── defaults
    │   └── main.yml
    ├── files
    ├── handlers
    ├── tasks
    │   └── main.yml
    └── templates
    └── motd.j2
```

Creating the role directory structure

- The **files** subdirectory contains **fixed-content** files and the **templates** subdirectory contains templates that can be deployed by the role when it is used.
- The other subdirectories can contain **main.yml** files that define default variable **values**, **handlers**, **tasks**, role **metadata**, or **variables**, depending on the subdirectory
- Handlers will contains a empty folder

```
# tree roles/

roles/
└── motd
├── defaults
│   └── main.yml
├── files
├── handlers
├── tasks
│   └── main.yml
└── templates
    └── motd.j2
```

Defining the role content

- Content of the Ansible **role** must be defined
- Start with ***ROLENAME*/tasks/main.yml** file
- **File** defines which ***modules*** to call on the managed hosts , role is applied
- Example:
  - The following **tasks/main.yml** file manages the **/etc/motd** file on managed hosts.
  - It uses the **template** module to copy the template named **motd.j2** to the managed host.
  - The template is retrieved from the **templates** subdirectory of the role.

```
cat roles/motd/tasks/main.yml

- name: deliver motd file
  template:
    src: templates/motd.j2
    dest: /etc/motd
    owner: root
    group: root
    mode: 0444
```

Using the role in a playbook

- To access a role, reference it in the **roles:** section of a playbook.
- Roles module, specify with role name called **motd**

```
cat use-motd-role.yml
---
- name: use motd role playbook
  hosts: remote.example.com
  roles:
    - role: motd
      system_owner: someone@host.example.com
```

Summary

- Roles organize Ansible tasks in a way that allows reuse and sharing.
- Role variables should be defined in **defaults/main.yml** when they will be used as parameters, otherwise they should be defined in **vars/main.yml**.
- Role dependencies can be defined in the **dependencies** section of the **meta/main.yml** file of a role.

- Tasks that are to be applied before and after roles are included with the **pre_tasks** and **post_tasks** tasks in a playbook.
- Ansible roles are referenced in playbooks in the **roles** section.
- Default role variables can be overwritten when a role is used in a playbook.

Lab
Implementing Roles

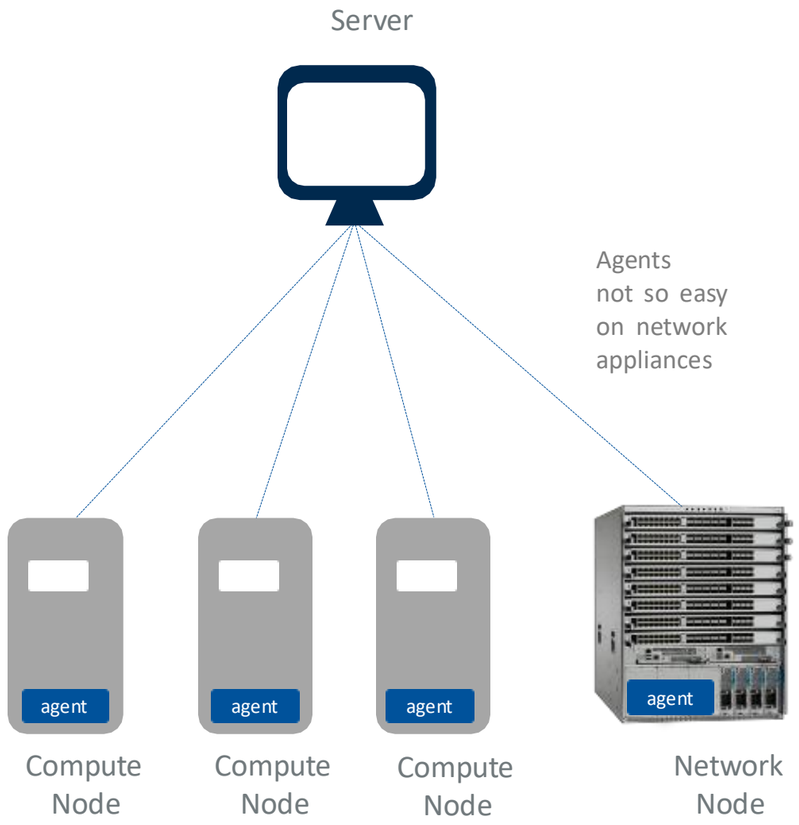
OneCloud Consulting

The Cloud Services Division of ePlus

Ansible Network Modules

Module 11

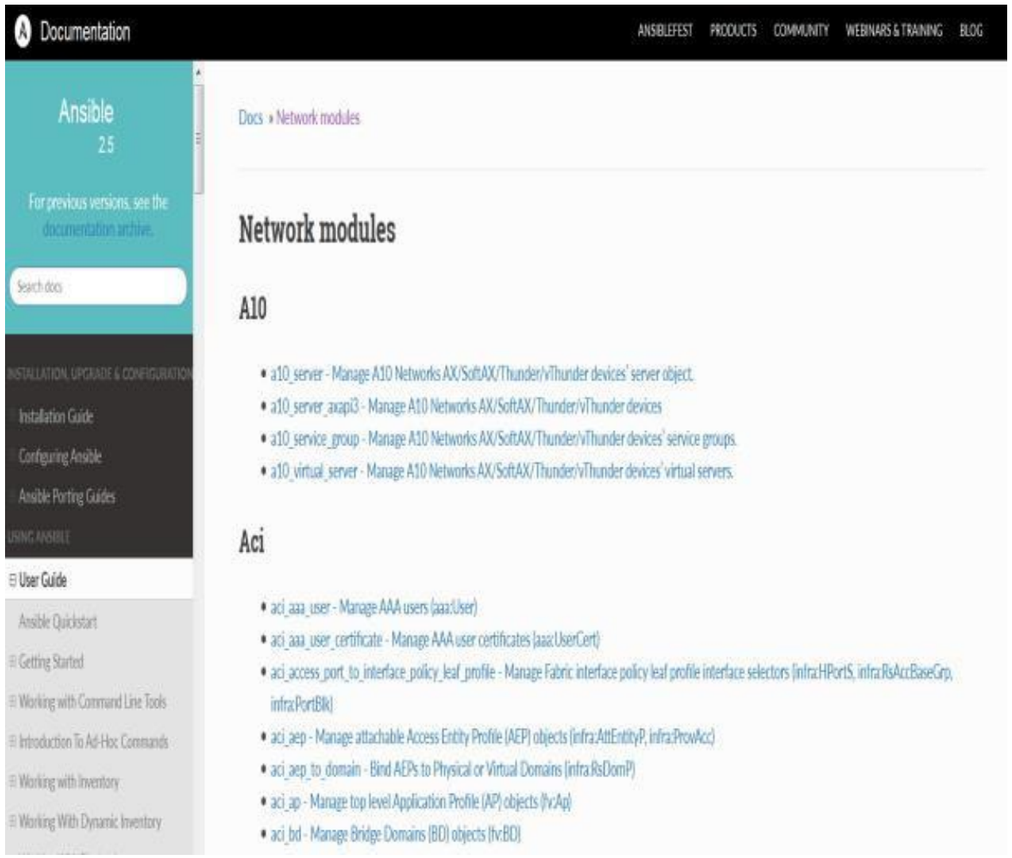
Overview



Why is Ansible the industry preferred platform for Network Device configuration management?

Agentless

Supported Network Modules



Source: https://docs.ansible.com/ansible/2.5/modules/list_of_network_modules.html

Summary

+ **Cisco**

  IOS, NXOS, ACI, IOS-XR, ASA

+ **Arista**

  EOS

Over 50+ network modules available today.

Common Network Tasks

- Show commands
- Configuration commands
- Save running config to startup config

Network Module “Show Command” Example

Ansible provides a simple yet powerful module for general show commands across any network device that doesn’t support Python.

Module	raw
Execution	ad-hoc
Use Cases	Getting output of CLI commands
	Getting output of CLI with grep
	Writing output of CLI to file

Network Module “Show Command” Example

Example ad-hoc commands using the Ansible raw module:

CLI

$ ansible all –m raw –a “show version” –u <username> –k

CLI with pipe include

$ ansible –i hosts ios –m raw –a “show ip traffic | include ttl” –u <username> –k

CLI with grep

$ ansible –i hosts eos –m raw –a “show interface | include drop” –u <username> –k

CLI output to file

$ ansible –i hosts nxos –m raw –a “show arp” –u <username> –k > output.txt

Cisco NX-OS Network Module Playbook Example

Show Command Module

```
-name: run multiple commands on remote
nodes
  nxos_command:
    commands:
      - show interfaces br
      - show mac address
      - show vpc
```

ConfigurationCcommand Module

```
- name: shutdown
  nxos_config:
  lines:
  -   shutdown
  parents: {{ item }}
  with_items:
        - interface Ethernet 1/1
        - interface Ethernet 1/2
```

Cisco IOS Network Module Playbook Example

Show Command Module

```
tasks:

  - name: run show version and check to see if
output contains IOS
    ios_command:
      commands: show version
      wait_for: result[0] contains IOS

- name: run show interface summary
  ios_command:
    commands: show interface summary
```

Configuration Command Module

```
tasks:

- name: Set CSR info
      set_fact:
      provider:
            host: "{{ CSRManagementIP }}"
            username: “{{ admin_user }}”
            ssh_keyfile: "{{ key_filename }}"

- name: make interface up
      ios_config:
        provider: "{{ provider }}"
        lines:
            - ip add 172.10.1.100 255.255.255.0
            - no shutdown
        parents: int gi2
```

Arista EOS Network Module Playbook Example

Show Command Module

```
- name: run multiple commands
  eos_command:
    commands:
      - show version
      - show interfaces
```

ConfigurationCcommand Module

```
- name: shutdown
  eos_config:
  lines:
  -   shutdown
  parents: {{ item }}
  with_items:
        - interface Ethernet1
        - interface Ethernet2
```

Real World Examples

Cisco NX-OS Playbook Example

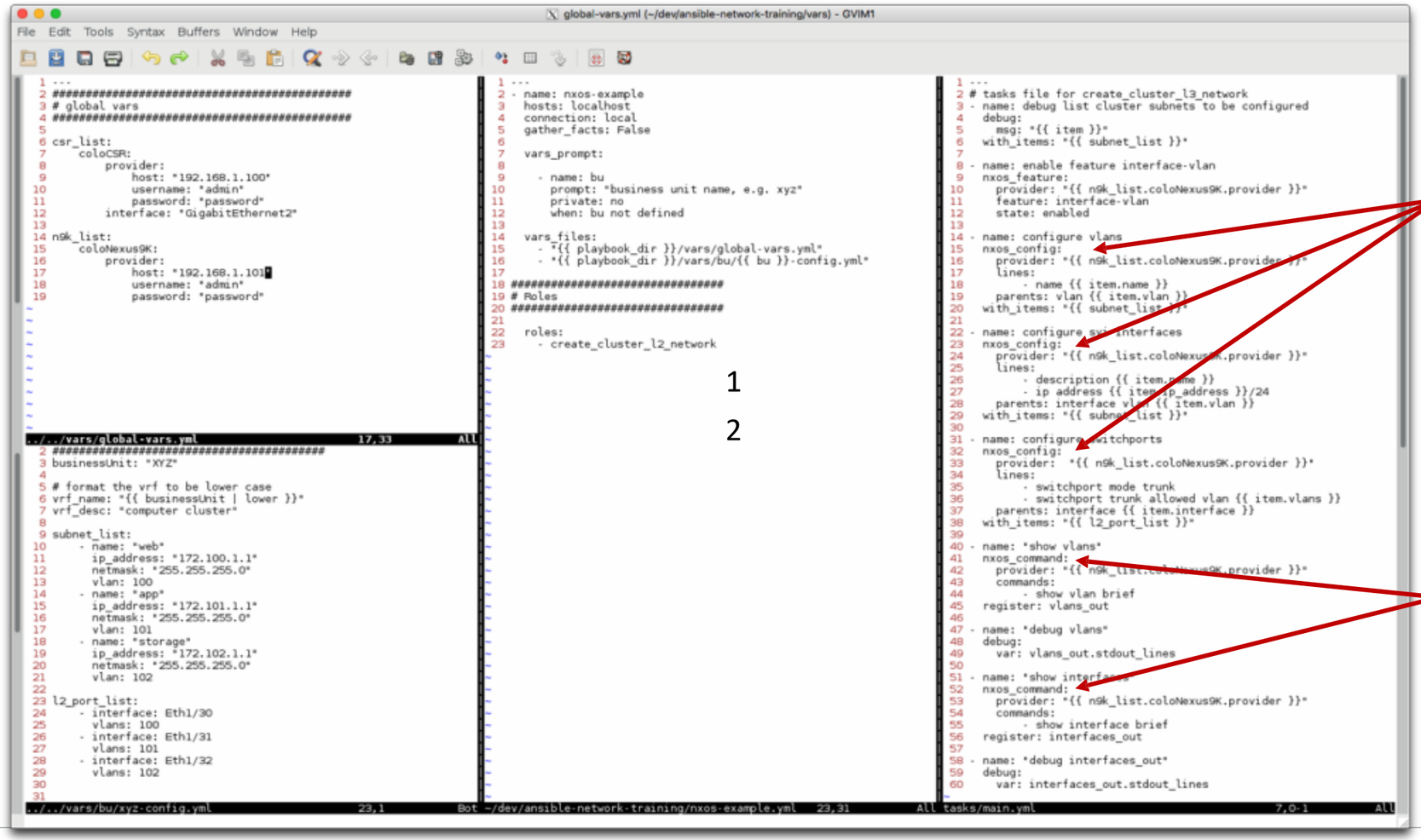
Network operator wants to:

+ Configure a Nexus 9K (nx-os)
+ Configure Vlans 100-102 and map them to switch ports eth 1/30-31
+ Configure SVI's with gateway IP's 172.**100**.1.1, 172.**101**.1.1, 172.**102**.1.1

1

1

Cisco NX-OS Playbook Example



Cisco NX-OS Playbook Example

Run the playbook

$ ansible-playbook –e “bu=xyz” nxos-example.yml

Cisco NX-OS Playbook Example

```
[N9k-Standalone-Pod-1#
[N9k-Standalone-Pod-1#
[N9k-Standalone-Pod-1#
[N9k-Standalone-Pod-1# show vlan

VLAN Name                              Status    Ports
---- --------------------------------- --------- -------------------------------
1    default                           active    Eth1/1, Eth1/4, Eth1/5, Eth1/6
                                                 Eth1/7, Eth1/8, Eth1/9, Eth1/10
                                                 Eth1/11, Eth1/12, Eth1/13
                                                 Eth1/14, Eth1/15, Eth1/16
                                                 Eth1/17, Eth1/18, Eth1/19
                                                 Eth1/20, Eth1/21, Eth1/22
                                                 Eth1/23, Eth1/24, Eth1/25
                                                 Eth1/26, Eth1/27, Eth1/28
                                                 Eth1/29, Eth1/33, Eth1/34
                                                 Eth1/35, Eth1/36, Eth1/37
                                                 Eth1/38, Eth1/39, Eth1/40
                                                 Eth1/41, Eth1/42, Eth1/43
                                                 Eth1/44, Eth1/45, Eth1/46
                                                 Eth1/47, Eth1/48, Eth1/49
                                                 Eth1/50, Eth1/51, Eth1/54
100  web                               active    Eth1/30
101  app                               active    Eth1/31
102  storage                           active    Eth1/32
1001 VLAN1001                          active

VLAN Type        Vlan-mode
---- ----- ----------
1    enet        CE
100  enet        CE
101  enet        CE
102  enet        CE
1001 enet        CE

Remote SPAN VLANs
------------------------------------------------------------------------------

Primary  Secondary  Type             Ports
-------  ---------  ---------------  -----------------------------------------
```

1

4

```
[N9k-Standalone-Pod-1#
[N9k-Standalone-Pod-1#
[N9k-Standalone-Pod-1#
[N9k-Standalone-Pod-1# show interface vlan 100
Vlan100 is down (Administratively down), line protocol is down, autostate enabled
  Hardware is EtherSVI, address is  188b.9de6.5a15
  Description: web  Internet Address is 172.100.1.1/24
  MTU 1500 bytes, BW 1000000 Kbit, DLY 10 usec,
   reliability 255/255, txload 1/255, rxload 1/255
  Encapsulation ARPA, loopback not set
  Keepalive not supported
  ARP type: ARPA
  Last clearing of "show interface" counters never
  L3 in Switched:
    ucast: 0 pkts, 0 bytes

[N9k-Standalone-Pod-1# show interface vlan 101
Vlan101 is down (Administratively down), line protocol is down, autostate enabled
  Hardware is EtherSVI, address is  188b.9de6.5a15
  Description: app  Internet Address is 172.101.1.1/24
  MTU 1500 bytes, BW 1000000 Kbit, DLY 10 usec,
   reliability 255/255, txload 1/255, rxload 1/255
  Encapsulation ARPA, loopback not set
  Keepalive not supported
  ARP type: ARPA
  Last clearing of "show interface" counters never
  L3 in Switched:
    ucast: 0 pkts, 0 bytes

[N9k-Standalone-Pod-1# show interface vlan 102
Vlan102 is down (Administratively down), line protocol is down, autostate enabled
  Hardware is EtherSVI, address is  188b.9de6.5a15
  Description: storage  Internet Address is 172.102.1.1/24
  MTU 1500 bytes, BW 1000000 Kbit, DLY 10 usec,
   reliability 255/255, txload 1/255, rxload 1/255
  Encapsulation ARPA, loopback not set
  Keepalive not supported
  ARP type: ARPA
  Last clearing of "show interface" counters never
  L3 in Switched:
    ucast: 0 pkts, 0 bytes
```

Here’s another example…

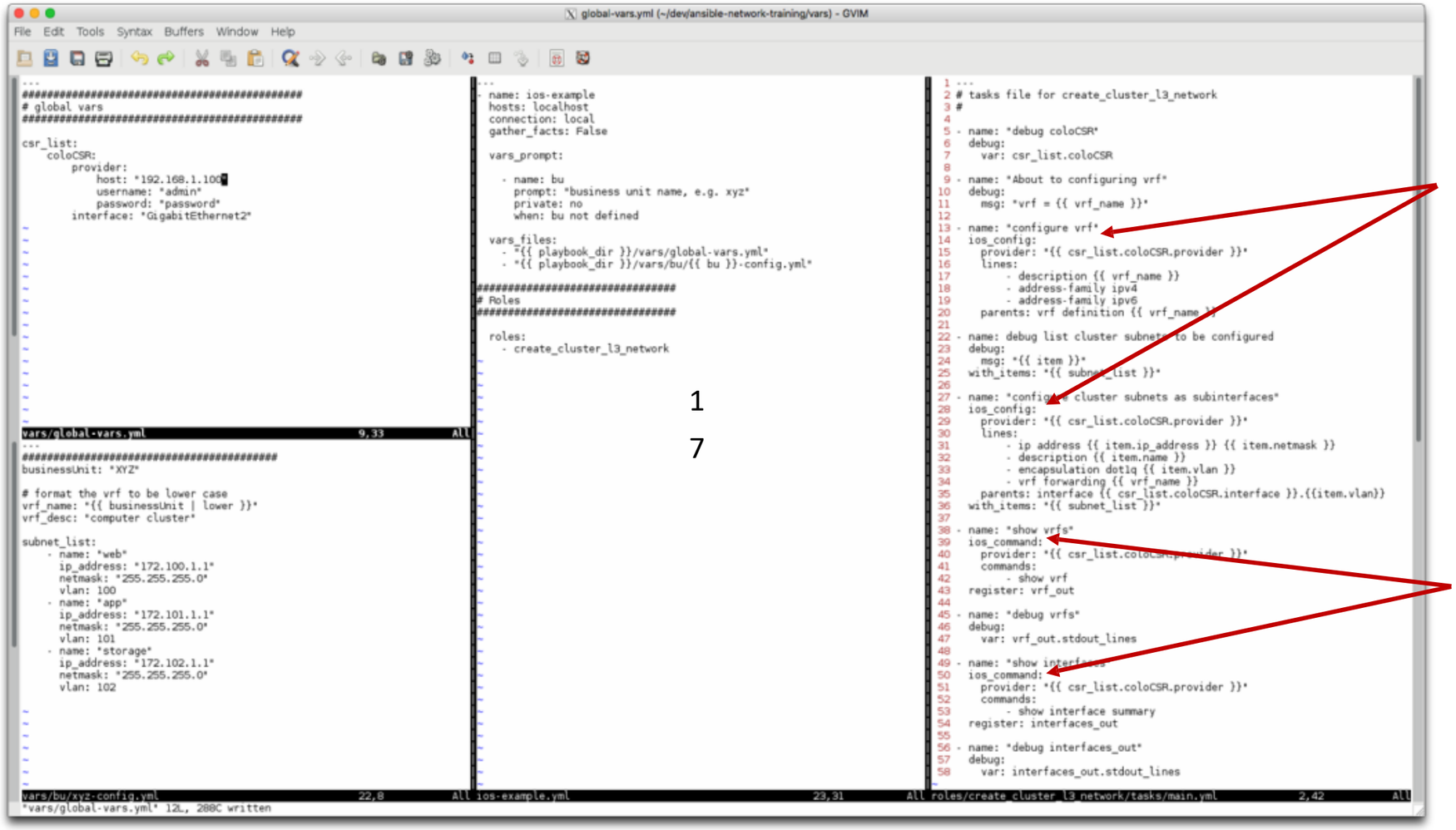
Cisco IOS Playbook Example

Network operator wants to configure networking for a new compute cluster. The cluster subnets will be managed by a specific division, xyz:

- **Web**
  - gateway IP 172.100.1.1/24
  - vlan 100
- **App**
  - gateway ip 172.101.1.1/24
  - vlan 101
- **Storage**
  - gateway ip 172.102.1.1/24
  - vlan 102

Use VRF ‘xyz’ and physical port GigabitEthernet2 for forwarding traffic

Cisco IOS Playbook Example



1

7

Cisco NX-OS Playbook Example

Run the playbook

```
$ ansible-playbook –e “bu=xyz” ios-example.yml
```

Cisco NX-OS Playbook Example

```
[csr1000v#show vrf
  Name                             Default RD          Protocols   Interfaces
  xyz                              <not set>           ipv4,ipv6   Gi2.100
                                                                   Gi2.101
                                                                   Gi2.102

[csr1000v#show interface summary

 *: interface is up
 IHQ: pkts in input hold queue     IQD: pkts dropped from input queue
 OHQ: pkts in output hold queue    OQD: pkts dropped from output queue
 RXBS: rx rate (bits/sec)          RXPS: rx rate (pkts/sec)
 TXBS: tx rate (bits/sec)          TXPS: tx rate (pkts/sec)
 TRTL: throttle count

  Interface                   IHQ       IQD       OHQ       OQD      RXBS      RXPS      TXBS      TXPS      TRTL
------------------------------------------------------------------------------------------------------------------
* GigabitEthernet1              0        71         0       655      4000         6         0         0         0
  GigabitEthernet2              0         0         0         0         0         0         0         0         0
  GigabitEthernet2.100          -         -         -         -         -         -         -         -         -
  GigabitEthernet2.101          -         -         -         -         -         -         -         -         -
  GigabitEthernet2.102          -         -         -         -         -         -         -         -         -
* GigabitEthernet3              0        35         0    738916 458458000    714188  94520000     89296         0
  VirtualPortGroup0             0         0         0         0         0         0         0         0         0
NOTE:No separate counters are maintained for subinterfaces
     Hence Details of subinterface are not shown
csr1000v#
```

Lab 11

End To End Orchestration Lab

OneCloud is a global provider of Cisco Technology education services and IT Consulting. We offer Instructor-led classroom Trainings, Next generation remote lab access and Technology Consulting.

Our vision

To be a leading Consulting and Technology Education Services firm recognized for our ability to implement innovative solutions for clients, by transforming best practice concepts into effective capabilities that build organizational excellence.