Avi Networks Ansible Documentation

Release 17.1.0

Avi Networks

Apr 25, 2020

Contents

CHAPTER 1

What is Ansible?

Ansible is an IT automation tool based on Python. It can managae network devices, configure systems, orchestrate cloud environments and deploy software.

It's goal is to create a secure way to manage infrastructure (using OpenSSH) and allow easy auditing by even those not familiar with the product. There's no complicated syntax to learn, or any complicated languages. It's designed for developers, system adminstrators, network engineers, security engineers, and release engineers.

Ansible is an agentless automation tool. Which means there is no agent that is required to be installed on the remote host. It just requires a "control" machine which executes Ansible playbooks against remote servers. Of course the control host will require access to the remote servers.

Avi Networks recognizes how automation using Ansible is drastically changing the way network and operations teams work and have decided to support the project by providing modules, and roles that allow automation in Ansible to be used against Avi Networks software.

CHAPTER 2

Our Automation

At Avi automation is in our blood. We understand how important automating the entire CI/CD pipeline is, and infact we use that same philosophy in our Ansible module development. Each module is built 1:1 from our APIs. Using automation we are able to build our modules from our objects automatically.

What does that mean?

Ever new feature, every new release will have modules that same moment to support those new features. No more waiting for a developer to write new modules to support the feature.

2.1 Introduction

Let's get started with the basics to Ansible. We will cover installation, getting started, understanding yaml, and the other key properties that are needed to use Ansible's vast amount of features.

2.1.1 Installation

Chosing a Version

We recommend using the latest major release available. Our playbooks, roles, and modules have been developed on 2.2, so we would recommend that version or later. However, 2.1 can be used but may have occasional issues on some of our playbooks, roles, and modules.

Requirements

Ansible can be ran from any machine running Python 2.6 or 2.7. In Ansible 2.2, a preview version of Python 3 support was added.

> **Warning:** Some modules and plugins will have additional requirements. Some may be libraries on the local machine, and some on remote machines. Please check the module documentation prior to executing the module.

For further installation requirements please visit Ansible's Installation documentation at http://docs.ansible.com/ansible/intro_installation.html

Installing Ansible

Ansible is available through many package repositories including EPEL, Apt, YUM, Portage, and many more. Here I will cover the most common ways to deploy Ansible to a host.

Installing via YUM

RPMs for Ansible are located in the EPEL repository. You will need to enable this repository on your host. To install EPEL on most recent operating systems use the following command.

```
sudo yum install epel-release
```

This will install the EPEL repository form your base OS repository. If this fails, you can also try and use the following command. Please modify <version> with the correct version of your distribution.

```
sudo rpm -ivh https://dl.fedoraproject.org/pub/epel/epel-release-latest-<version>.
↪noarch.rpm
```

Once EPEL is installed you will be able to simply run the following command, which will install Ansible from EPEL onto your host.

```
sudo yum install ansible
```

Installing via Apt

Ansible is not included in any of Ubuntu's repositories, and needs to be installed via PPA. To use PPA you will need to run the following commands. These commands will install the proper package tools to add the PPA repository, and update your cache, and then install the Ansible package.

```
sudo apt-get install software-properties-common
sudo apt-add-repository ppa:ansible/ansible
sudo apt-get update
sudo apt-get install ansible
```

Installing via Pip

Ansible can also be installed via Pip. Pip is a Python package manager. If it is not already available on your system you may install it with the following command.

```
sudo easy_install pip
```

Then to install Ansible the following command.

```
sudo pip install ansible
```

It has been documented that on OS X Mavericks there could be issues with the compiler. To work around the issue use the following command.

```
sudo CFLAGS=-Qunused-arguments CPPFLAGS=-Qunused-arguments pip install ansible
```

Installing via Brew

Mac OSX has an awesome utility called brew, which is available if Homebrew is installed. If installed, to install ansible becomes one simple command.

```
brew install ansible
```

2.1.2 Getting Started

How does Ansible Work?

Ansible uses OpenSSH by default to connect and execute commands on remote hosts. This also allows the user to take advantage of ControlPersist, Kerberos, and the options avaialble in the users SSH config (~/.ssh/config). If OpenSSH is too old on the host, and it requires features in a newer version, Ansible will attempt to use paramiko to execute. If you want to use Kerberos in this situation, please upgrade your OpenSSH version.

By default, Ansible will attempt to use your SSH key to authenticate the current user to the remote server. To prevent this and customize these options please review the following:

`--ask-pass` option to allow us to supply a password for login to the remote host via prompt.

`--ask-become-pass` option to allow us to supply a password for sudo/become login to the remote host via prompt

`--user <myuser>` option to allow us to supply a user for login to the remote host

Host Key Checking

Host checking is used to verify the known host key against the host key on the remote server. Host keys are often times located in your `known_hosts` file. This file is usually located at `~/.ssh/known_hosts`. If this file doesn't have a host key for the host it will ask if you trust it. If not then it proceeds. However, if the host is in there, but the key doesn't match, it will return an error. When using automation sometimes this is common, and not an issue. In Ansible we have a way to disable errors on host key checking. You can edit `/etc/ansible/ansible.cfg` or `~/.ansible.cfg` and set `host_key_checking = False`. For example:

```
[defaults]
host_key_checking = False
```

You can also use an environment variable, however I do discourage this practice unless required.

```
export ANSIBLE_HOST_KEY_CHECKING=False
```

Testing Connectivity

When using Ansible, we rely on connectivity to the remote host via ssh. To test if we can connect we can use

```
ansible 10.20.10.200 -m ping -u root --ask-pass
```

This command will connect over SSH to the remote host and verify connectivity. We will explain later more about this type of Ansible command.

2.1.3 Understanding YAML

This page will explain the structure of YAML (Yet syntax which is the syntax for Ansible playbooks. Ansible chose YAML because of it's ease of use and readability. It also maintans the same ability JSON does to display objects.

YAML Basics

Indentation

YAML uses a fixed indentation scheme. Each level consists of 2 spaces, do not use tabs.

Key-Value Pairs

In YAML we display dictionaries as key-value pairs. Also know as hashes. In YAML, dictionary keys are represented by strings terminated by a colon. Following a space the value is specified. For example:

```
key: value
```

In json it would convert to, which just so happens to be the same as a json dictionary.

```
{ 'key': 'value'}
```

Nesting Dictionaries

To nest dictionaries in YAML we would use the following syntax.

```
first_level:
  second_level_key: second_level_value
```

In python this would be recognized as:

```
{
"first_level": {
  "second_level_key": "second_level_value"
  }
}
```

Arrays

In YAML we also can create arrays. Arrays are basically lists of items. Arrays can be a list of dictionaries or single items. Each item in a list is prefixed with a − and then a space.

```
- item1
- item2
- item3
```

This is a simple list of 3 items.

Often times in YAML we will display an array as a value of a key.

```
items:
  - item1
  - item2
  - item3
```

Which in python would be seen as

```
{ "items": ["item1", "item2", "item3"] }
```

Handling Newlines

Sometimes we will want to make our YAML values span multiple lines, or maybe shorten a really long line into a smaller one. We can do that in two ways.

The symbol | will maintain your newlines in the value.

```
include_newlines: |
  I definitely needed some
  new lines in this output
```

Will render as:

```
I definitely needed some
new lines in this output
```

The symbol > will ignore newlines and move all lines into a single line of text.

```
ignore_newlines: >
  I am really
  starting to enjoy
  using YAML
```

Will render as:

```
I am really starting to enjoy using YAML
```

2.1.4 Inventory

Inventory File

Ansible uses an inventory file to provide a list of hosts to Ansible. These hosts can be seperated into groups, or individual in the inventory file. The default inventory file is located at `/etc/ansible/hosts` but you can supply your own and point to it using the `-i <inventory_file>` option when running an Ansible playbook or command. The inventory file is usually written in INI format.

Hosts

In the inventory file hosts by themselves are the simplest to define. A simple inventory file can look like this:

```
host.example.com
```

Groups

When using groups in your inventory file you can classify your hosts and decide what groups of hosts you want to control from Ansible. You can also include a single host in multiple groups. Such as the being a webserver and a database at the same time.

Groups are defined like this:

```
[webserver]
server1.example.com
server2.example.com

[database]
db1.example.com
db2.example.com
```

There is one catch though. When using groups we cannot place single hosts not included with any groups below. So the following will apply:

```
# my single hosts
host.example.com
host1.example.com
host2.example.com

# Put Groups below
##################
[webserver]
server1.example.com
server2.example.com

[database]
db1.example.com
db2.example.com
```

Non-standard SSH Ports

If you are running non-standard SSH ports on your hosts you can also specify the port on the hostname seperated by a colon. For example:

```
host.example.com:2222
```

Host Patterns

In an inventory file we can also define multiple hosts with one entry. You can specify numeric ranges, as well as alphabetic ranges.

Numeric range example:

```
host[1:10].example.com
```

Alphabetic range example:

```
server1[a:e].example.com
```

Host Entry Components

:: <alias> <special variables> <variables>

host alias: can be a hostname or just an alias, if using an alias, you will need to specify the special variable `ansible_host=10.20.20.10`

special variables: there are many of these, they include the *Behavioral Inventory Parameters*

variables: these are variables you want to define specifically to a host that would be used in your playbooks

Using Hosts not in Inventory

When looking to use hosts without an inventory file, we can specify the `ansible-playbook` or `ansible` ad-hoc command as

```
ansible-playbook -i hostname, playbook.yml
```

If you noticed we added the `,` after the hostname. This specifies to Ansible that we want to use a comma separated list of hosts not related to the hosts file.

Behavioral Inventory Parameters

These are also known commonly as *Behavioral Inventory Parameters* and can all be found here: http://docs.ansible.com/ansible/intro_inventory.html#list-of-behavioral-inventory-parameters

2.1.5 Dynamic Inventory

Dynamic inventory allows us to with a script dynamically pull in an inventory from an environment. It could query anything you have with an API and provide a list of servers, and groups of servers, it can even group servers by tags as well. For example, if you have a bunch of servers in AWS, using a Dynamic inventory, you can run an Ansible Playbook against servers based on their names, and tags. Helps when you have a dynamic environment itself.

With a dynamic inventory, you can also pull data from cloud providers, LDAP, cobbler, and other software that may be keeping track of your servers. Ansible can support all of these using it's external inventory system.

If you are interested in these you can read more, and view examples on the Ansible documentation website: http://docs.ansible.com/ansible/intro_dynamic_inventory.html

2.1.6 Patterns

When running Ad-Hoc commands Ansible can allso accept patterns, for example

```
ansible \*.example.com -m ping
```

This command would ping all hosts in your inventory file that contains .example.com at the end of the host or alias.

If you wanted to use all hosts you can use:

```
ansible all -m ping
```

You can allso use:

```
ansible \* -m ping
```

For more information please visit http://docs.ansible.com/ansible/intro_patterns.html

2.1.7 Ad-Hoc Commands

Ad-hoc commands are a great way to quickly issue a command against a host without running an entire playbook. They also don't require knowing YAML.

Ad-Hoc Command Structure

```
ansible <group or hostname> -m <module name> -u username [--ask-pass] [--become] [--
↪ask-become-pass]
```

For more options just type `ansible --help` and a full list of options will be available.

2.1.8 Configuration File

Ansible provides a configuration file that allows you to customize the way Ansible runs. Ansible provides a number of ways to provide the configuration file. They are also processed in the following order.

- `ANSIBLE_CONFIG` environment variable
- `ansible.cfg` located in the current working directory
- `.ansible.cfg` located in the users home directory
- `/etc/ansible/ansible.cfg` the main ansible configuration file

The configuration file, like the inventory file are in INI format. For most people the default file that was provided on installation should be enough. However, if you need to customize it please use http://docs.ansible.com/ansible/intro_configuration.html as a reference. There are many values that can be used to control Ansible.

2.2 Playbooks

2.2.1 Hosts

Each play will require a `hosts` line. This can be a host pattern, host group, host, and can be separated by colons. The section looks like this. `all` can be replaced with something like `*.example.com` as well.

```
- hosts: all
```

2.2.2 Users

In playbooks we also can specify which user we want to use for the play. To do this we will use the `remote_user` value. For example to use the user `centos` we would use this example.

```
- hosts: all
  remote_user: centos
```

This allows us to modify the remote user instead of using our current logged in user on the control machine (the machine we run Ansible from).

2.2.3 Priviledge Escalation

Often times we may try to make a change that would require sudo access. By default this priviledge isn't granted to Ansible. We will need to provide it. To do this we will of course need to provide your user sudo on the remote machine. This will have to be done manually, or you will need to run Ansible with a `root` or different account that has the access to modify your sudoers priviledge.

Once the remote machine is configured properly we have a series of directives that we can use.

become when set to `true` or `yes` will escalate the priviledge

become_user choose which user will be used to escalate priviledge

become_method overrides the default in your ansible configuration file, can use `sudo`, `su`, `pbrun`, `pfexec`, `doas`, `dzdo`, `ksu`

Below is an example of using priviledge escalation:

```
- hosts: all
  remote_user: centos
  become: yes
```

We can also apply this to specific tasks throughout the play too!

```
- hosts: all
  remote_user: centos
  tasks:
    - service: name=firewalld status=stopped enabled=no
      become: yes
```

We can also use other methods like `su` to escalate priviledge.

```
- hosts: all
  remote_user: centos
  become: yes
  become_method: su
```

When using become method and not using a SSH key, you will need to provide the `ansible-playbook` command the password for priviledge escalation. To do this you can use the option `--ask-become-pass`, followed by the password. If you run the playbook and it hangs it's possibly stuck at the priviledge escalation prompt, you can use *Control-C* to quit and then try again with a valid password.

2.2.4 Tasks

Playbooks also will include a list of tasks. Each are run in the order they are listed. Tasks can also include vars specific to the task itself, as well as specific arguments documented in the module.

Tasks will look like this:

```
tasks:
  - name: Stop the Firewalld service and disable it from boot
    service: name=firewalld status=stopped enabled=no
```

We can also specify the task like this:

```
tasks:
  - name: Stop the Firewalld service and disable it from boot
    service:
      name: firewalld
      status: stopped
      enabled: no
```

It's really up to you, however the first is usually cleaner on some modules, while the second can be useful for modules with many values. The second will also use YAML for everything, the first will likely need specific json formating for complex values.

If a task fails please keep in mind the playbook will stop. You will need to fix the task, then you will need to rerun the playbook. Because of this idempotency is extremely important. If you do not ensure idempotency of your tasks you will possibly run the same command twice.

When using `shell` or `command` modules they will run the command again. To prevent this you should use a `creates` flag or use `when` and have a previous task register if the task needs to run again.

Note: `command` and `shell` modules are the only modules that do not follow key=value format. They are in the free form format of `shell: cat myfile` or `command: cat myfile`.

You can also ignore errors if your command task results in a `1` or if a module fails. To ignore errors simply add `ignore_errors: True` to your task.

```
tasks:
  - name: get contents of myfile
    shell: cat myfile
    ignore_errors: True
```

You can also use previously defined variables in your tasks.

```
vars:
  filename: myfile
tasks:
  - name: get contents of {{ filename }}
    shell: cat {{ filename }}
```

2.2.5 Handlers

Ansible also has an event system which allows tasks to trigger actions. To take advantage of this we have "Handlers". Handlers can be called using the `notify` option on tasks. A nice benefit to this is when you have multiple files that when edited need to restart a service, will notify the hander task which will signal it to run at the end of a play. If multiple files need to restart the same service, it will only restart the service once at the end of the play (instead of multiple times). An example of this is below:

```
handlers:
  - name: restart service
    service: name=service state=restarted
tasks:
  - name: modify config file
    template: src=config.j2 dest=/etc/config.conf
    notify: restart service
```

This will tell Ansible that at the end of the play it will restart the service.

For more information on handlers please visit: http://docs.ansible.com/ansible/playbooks_intro.html#handlers-running-operations-on-change

2.2.6 Roles

Roles are a component of Ansible that allow you to reuse tasks, and other components by putting them in a role. Which can be distributed to other people via Ansible Galaxy, or shared internally to allow reusing sets of tasks, vars, etc, to deploy your applications.

An example role can look like this:

```
---
- hosts: controllers
  roles:
    - role: avinetworks.avicontroller
      con_controller_ip: 10.10.27.101
      con_cores: 4
      con_memory_gb: 12
```

To explain this playbook we will show we have a role: `avinetworks.avicontroller`, which has variables `con_controller_ip` and `con_cores`, and `con_memory_gb` specified. There are many others possible, but since we are just evaluating the example we will use this. The variables are then passed into the roll to replace any defaults, or simply provide variables that require values. These are referred by roles as "Role Variables", and lists of possible options are usually in the documentation of the role README for example: https://galaxy.ansible.com/avinetworks/avicontroller/ click on the "README" tab of the Galaxy Role.

2.2.7 Include

Play Include

There are two types of includes in Ansible. There are Play includes, and Task includes. Play includes will include other plays in your playbook. For example if we have a playbook `playbook1.yml` and we want to include that playbook in another playbook, such as `master_play.yml`, `master_play.yml` would look like this.

```
---
- include: playbook1.yml

- name: Master play playbook
  hosts: all
  tasks:
    - debug: mg="Extra Task"
```

This playbook will execute everything in `playbook1.yml` and then will continue with the debug task in the next play in the `master_play.yml`.

Task Include

The second type of include is Task include. Task includes are used to include other files with tasks in them, and can help break one giant set of tasks into others, as well as control when the other tasks are ran, such as a group of tasks you only want ran when a specific condition is met.

For example, here is Ubuntu.yml, this file has a few tasks specific to Ubuntu distributions. If you notice we don't need `tasks:` at the top of the file.

```
- name: Docker | CE | APT | Add Docker GPG Key
  apt_key:
    id: 0EBFCD88
    url: https://download.docker.com/linux/ubuntu/gpg
    state: present

- name: Docker | CE | APT | Configure Docker repository
  apt_repository:
    repo: "deb [arch=amd64] https://download.docker.com/linux/ubuntu {{ ansible_
↪distribution_release }} stable"
    state: present

- name: Docker | CE | APT | Enable Edge repository
  apt_repository:
    repo: "deb [arch=amd64] https://download.docker.com/linux/ubuntu {{ ansible_
↪distribution_release }} edge"
    state: present
  when: docker_channel == "edge"
  notify: Docker | CE | APT | Upgrade to Edge
```

Because these use APT and the repo is for Ubuntu we only need these to run on Ubuntu. So here's how we would include this in the playbook as a task include.

```
- name: Master play playbook
  hosts: all
  tasks:
    - name: Docker | CE | APT | Ubuntu
      include: Ubuntu.yml
      when: ansible_distribution == "Ubuntu"
```

This can make complex task executions much easier and faster as if all the tasks in `Ubuntu.yml` don't need to run (system doesn't match as Ubuntu) then it will skip the entire set of tasks in `Ubuntu.yml`.

You can also reuse includes and change variables in them. For example lets create `message.yml`:

```
- name: Give a message
  debug: msg={{ message }}
```

Let's call this in the master playbook and change the message.

```
- name: Master play playbook
  hosts: all
  tasks:
    - include: message.yml message="This is my message to you"
    - include: message.yml message="This is my second message to you"
```

Static and Dynamic Includes

In the previous examples I covered static includes. But now in Ansible 2.0 and later we have Dynamic includes, which allow us to use variables in our includes. For example:

```
- name: Master play playbook
  hosts: all
  tasks:
    - include: message.yml message={{ item }}
      with_items:
        - This is my message to you
        - This is my second message to you
```

We can also use other variables in a dynamic include. Previously we did this:

```
- name: Master play playbook
  hosts: all
  tasks:
    - name: Docker | CE | APT | Ubuntu
      include: Ubuntu.yml
      when: ansible_distribution == "Ubuntu"
```

When we wanted the Ubuntu file to run when the OS matched. Now we can take this a step farther. Let's say we have multiple operating systems, Ubuntu, CentOS, RedHat, etc. We can create task files for each of those, then use the following to dynamically select which one we want to run.

```
- name: Master play playbook
  hosts: all
  tasks:
    - name: Docker | CE | Repo
      include: "{{ ansible_distribution }}.yml"
```

For more information regarding Dynamic and Static includes please visit: http://docs.ansible.com/ansible/playbooks_roles.html#dynamic-versus-static-includes

2.2.8 Executing Playbooks

To execute a playbook, you can simply follow this format:

```
ansible-playbook playbook.yml
```

There are many options that can be used alongside the ansible-playbook command. To view these please use `ansible-playbook --help`.

Passing Variables

When executing the playbook command, we can also pass variables via the command line. To do so we will use the following command:

```
ansible-playbook playbook.yml --extra-vars "variable=value variable2=value2"
```

Checking Syntax

The `ansible-playbook` command can also be used to validate the syntax of your playbook without executing it against the remote hosts. This will help prevent errors from causing mid play crashes and other issues. To do this use the following command.

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

This is extremely useful as a way to lint test your playbooks.

2.2.9 Variables

Using Ansible we can also use variables to use the same playbooks, plays, tasks, etc. We can also create variables by registering the results from previous tasks. Variables can also be used in Conditionals and Loops.

Rules of Variables

Variables must:

- start with a letter
- not include a –, a space, or .
- not be a number

Examples of invalid variables are `123`, `variable-name`, `variable name`, and `variable.name`.

Valid ones can be `variable`, `variable_name`, and `variable1`

You can also use dictionaries which are supported to map keys to values.

```
variable:
  key1: value
  key2: value
```

The previous variable can be referenced by either `variable['key1']` or by `variable.key1` however you cannot define them in that way, this only works one direction.

Using "dot" notation the following are reserved as they are python methods:

`add`, `append`, `as_integer_ratio`, `bit_length`, `capitalize`, `center`, `clear`, `conjugate`, `copy`, `count`, `decode`, `denominator`, `difference`, `difference_update`, `discard`, `encode`, `endswith`, `expandtabs`, `extend`, `find`, `format`, `fromhex`, `fromkeys`, `get`, `has_key`, `hex`, `imag`, `index`, `insert`, `intersection`, `intersection_update`, `isalnum`, `isalpha`, `isdecimal`, `isdigit`, `isdisjoint`, `is_integer`, `islower`, `isnumeric`, `isspace`, `issubset`, `issuperset`, `istitle`, `isupper`, `items`, `iteritems`, `iterkeys`, `itervalues`, `join`, `keys`, `ljust`, `lower`, `lstrip`, `numerator`, `partition`, `pop`, `popitem`, `real`, `remove`, `replace`, `reverse`, `rfind`, `rindex`, `rjust`, `rpartition`, `rsplit`, `rstrip`, `setdefault`, `sort`, `split`, `splitlines`, `startswith`, `strip`, `swapcase`, `symmetric_difference`, `symmetric_difference_update`, `title`, `translate`, `union`, `update`, `upper`, `values`, `viewitems`, `viewkeys`, `viewvalues`, `zfill`

Inventory Variables

Host vars

To specify variables in your inventory files for a specific host, you would use the following `key=value` format.

```
host1 key1=value1 key2=value2
```

This will provide the host with `key1` and `key2`, which will be used in the playbooks and ad-hoc commands referencing this host.

Group vars

To specify variables in your inventory files you would put the variable on the same line in `key=value` format.

To specify group variables in an inventory file:

```
[mygroup]
host1
host2
```

(continues on next page)

(continued from previous page)

```
[mygroup:vars]
variable1=value1
variable2=value2
variable3=value3
```

This would then provide each of these 3 variables for all hosts in `mygroup`.

Playbook Variables

In playbooks we can define variables in plays by the following.

```
- hosts: all
  vars:
    variable1: value1
```

Included files and roles

We've already covered this previously. To specify a variable to an include:

```
tasks:
  - include: tasks.yml variable1=value
```

You can also specify variables this way as well.

```
tasks:
  - include: tasks.yml
    vars:
      variable1: value
```

To use the value in the tasks.yml file we will reference the var as `{{ variable1 }}`.

Registered Variables

An extremely useful feature of Ansible is the ability to register the output of a task into a variable so that it can be referenced later. To view possible output of a task that would be in the registered variable, you can look at the output of `-v`. What is included in the `results` value is what would be contained in the registered variable.

For example:

```
- hosts: all
  tasks:
    - stat: path=/tmp
      register: tmp_folder_data

    - debug: msg={{ tmp_folder_data }}
```

This sniplet would look for `/tmp` on the remote host, and get the information of that folder as per the `stat` module, and then provide us with all the information of that folder by the debug module and printing it to output.

Accessing Variable Data

Sometimes our variables may have more data to them than just a single value. For example the previous example of using `stat` module. It returned a bunch of information to us.

```
{
    "tmp_data": {
        "changed": false,
        "stat": {
            "atime": 1481748353.0,
            "ctime": 1492640380.9926686,
            "dev": 1,
            "executable": true,
            "exists": true,
            "gid": 0,
            "gr_name": "root",
            "inode": 281474977014021,
            "isblk": false,
            "ischr": false,
            "isdir": true,
            "isfifo": false,
            "isgid": false,
            "islnk": false,
            "isreg": false,
            "issock": false,
            "isuid": false,
            "mode": "1777",
            "mtime": 1492640380.9926686,
            "nlink": 2,
            "path": "/tmp",
            "pw_name": "root",
            "readable": true,
            "rgrp": true,
            "roth": true,
            "rusr": true,
            "size": 0,
            "uid": 0,
            "wgrp": true,
            "woth": true,
            "writeable": true,
            "wusr": true,
            "xgrp": true,
            "xoth": true,
            "xusr": true
        }
    }
}
```


To access a specific item for example `exists`, in this object we can use two types of notation.

```
{{ tmp_data["stat"]["exists"] }}
```

```
{{ tmp_data.stat.exists }}
```

Both will return `true` as the result.

To access the first element of an array we would use `data[0]`.

Variable Precedence

Because of how many possible places we can put a variable, we will need to understand variable precedence. Top of the list is the weakest, bottom is the strongest.

- role defaults
- inventory INI or script group vars
- inventory group_vars/all
- playbook group_vars/all
- inventory group_vars/*
- playbook group_vars/*
- inventory INI or script host vars
- inventory host_vars/*
- playbook host_vars/*
- host facts
- play vars
- play vars_prompt
- play vars_files
- role vars (defined in role/vars/main.yml)
- block vars (only for tasks in block)
- task vars (only for the task)
- role (and include_role) params
- include params
- include_vars
- set_facts / registered vars
- extra vars (always win precedence)

Extra vars are what we specify on the command line as we talked about earlier with `-e` or `--extra-vars`.

There are also 3 types of variable scopes, *Global*, *Play*, and *Host*.

- Global is set via command line, Environment Variable, or using the config file.
- Play is set in the play, using vars entries, include_vars, role defaults, and vars.
- Host is set in the inventory, facts, or registered output from tasks.

Ansible Facts

Ansible by default will gather facts about the remote host. You can see all the facts gathered from a remote host by using the command:

```
ansible hostname -m setup
```

Of course replace hostname with the name of the host, the host will need to be in your inventory file. Once it runs it will return a JSON object with all the information Ansible knows of the host. It can return interface information, disk information, kernel information, OS information, and much more.

To turn off Ansible Facts on a host you would use the following:

```
- hosts: all
  gather_facts: no
```

Setting `gather_facts` to `no` will disable the gathering of facts from the remote host.

Ansible also has Local Facts, which can be provided by custom facts modules. For more information please visit: http://docs.ansible.com/ansible/playbooks_variables.html#local-facts-facts-d

Using Variables in Jinja Templates

Ansible uses the Jinja template system to create files and handle variables within playbooks. An example of a a template task and the jinja template would be:

```
vars:
  server_port: 9060
  server_ip: 192.168.1.20
tasks:
  - template: src=server.j2 dest=/etc/app/server.conf mode=0644
```

```
port={{ server_port }}
serverIP={{ server_ip }}
```

So the end result of the file located at /etc/app/server.conf would be:

```
port=9060
serverIP=192.168.1.20
```

Using Variables in Tasks

Ansible allows us to use Jinja within playbooks as well. Making reusing tasks much easier as well as customizing tasks for a different operating system, or any configuration that may differ from server to server.

For example we can change the variables based on the os distribution. Then use those to define a package name. This allows you to support cases in which Apache on CentOS is `httpd` but on Ubuntu is `apache`. We can load the variables specific to that OS and use those.

Ubuntu.yml

```
package_name: apache
```

CentOS.yml

```
package_name: httpd
```

Playbook Excerpt

```
- name: Include OS Specific Variables
  include_vars: "{{ ansible_distribution }}.yml"
- name: Install Package
  package: name={{ package_name }} state=present
```

If you didn't notice, when we did the `include_vars` the value had `""` (double quotes) around it. Any value that starts with a variable will need quotes around it. This is a YAML syntax usage correction. Failure to do this will cause Ansible to hit an error on execution.

Jinja Filters

There are many filters that can be extremely useful in modifying playbooks, values, and even dynamically handling data for variables. We can force things to be uppercase, lowercase, combine items, and much more. Jinja has a list of built-in filters documented here: http://jinja.pocoo.org/docs/2.9/templates/#builtin-filters

We will go over a few of these filters that have been common throughout our experience, and provide you some examples.

Math Operators

Jinja will also let us perform mathmatical actions on values. For example

```
- hosts: all
  vars:
    some_number: 2
  tasks:
    - debug: msg={{ some_number + 1 }}
```

The result of this would give us a message with the number `3`.

+ Adds objects together, it's not recommended to use this for strings, for strings use ~ which will concatenate strings.

- Will subtract the second number from the first

/ Divides two numbers and will return a float

// Divides two numbers and will return a truncated integer, this does not round, it just drops everything after the .

% Provides the remainder of an integer division

***** Multiplies the left operand with the right. `{{ 2 * 4 }}` will return `8`. `{{ '#' * 40 }}` would return 40 # symbols

****** Raises the left operated to the power of the right.

Comparisons

== Compares two objects for equality.

!= Compares two objects for inequality.

> true if the left hand side is greater than the right hand side.

>= true if the left hand side is greater or equal to the right hand side.

< true if the left hand side is lower than the right hand side.

<= true if the left hand side is lower or equal to the right hand side.

These are extremely common in `when` portions of tasks.

Logic

and Return true if the left and the right operand are true.

or Return true if the left or the right operand are true.

not negate a statement (see below).

(expr) group an expression.

These can be useful when handing `when` statements or `if` statements in your jinja templates.

Other Operators

The following operators are very useful but don't fit into any of the other two categories:

in Perform a sequence / mapping containment test. Returns true if the left operand is contained in the right. {{ 1 in [1, 2, 3] }} would, for example, return true.

is Performs a test.

Applies a filter.

~ Converts all operands into strings and concatenates them. `{{ "Hello " ~ name ~ "!" }}` would return (assuming name is set to 'John') `Hello John!`.

() Call a callable: {{ post.render() }}. Inside of the parentheses you can use positional arguments and keyword arguments like in Python: `{{ post.render(user, full=true) }}`.

. / [] Get an attribute of an object.

ipaddr

For instance lets validate the ip address we pass as a variable.

To use `ipaddr` we will need to install `netaddr`.

```
pip install netaddr
```

```
ansible-playbook playbook.yml --extra-vars "controller_ip=10.23.222.10"
```

```
- hosts: any
  roles:
    - role: avinetworks.avicontroller
      con_controller_ip: {{ controller_ip | ipaddr }}
```

If the supplied `controller_ip` isn't a valid IP, the value of `con_controller_ip` will be "False" which would result in a failure of the execution.

default

The `default()` filter allows us to provide a default to a variable if it's not defined. Preventing an error if a value isn't provided. For example:

```
{{ my_string|default('You didn't provide a string')}}
```

Would provide the result `You didn't provide a string` if `my_string` wasn't defined.

In Ansible Use

```
tasks:
  - debug: msg={{ my_string | default('You didn't provide a string')}}
```

When executing this if you don't provide a variable named `my_string` then the `debug` module will return the message `You didn't provide a string`. This can be useful when not requiring variables.

join

When using the `join()` filter we can join items in an array into a string.

```
vars:
  my_list:
    - item1
    - item2
    - item3
tasks:
  - debug: msg={{ my_list | join(',')}}
```

Would create me a comma seperated list of `my_list` which would look like `item1,item2,item3`

Python Methods

In practice we found that Python methods can also be useful when parsing text that is put into a variable, replace text, and many more python methods.

split

For instance, we want to split up a comma seperated list that was provided as a string variable to Ansible.

To do this we can do the following using the `split` python method.

```
tasks:
  - name: build server list
    set_fact:
      servers: "{{ servers|default([]) + [{'ip': {'addr': item, 'type': 'V4'}}] }}"
    with_items: "{{ pool_servers.split(',') }}"
```

Using that task we are able to take in a comma seperated list of server ip addresses as `pool_servers` and iterate through those and append those to the servers variable.

2.2.10 Loops

Loops are extremely common in playbooks and tasks. It's a great way to reuse a single task to iterate through a list of values or objects. We can use it to install multiple packages, maybe run a task on multiple objects.

Standard Loops

The most common loop you will likely use is the standard loop. Which we use `with_items` provided to the task.

```
- name: Install packages
  package: name={{ item }} state=present
  with_items:
    - httpd
    - memcached
```

This will run the task twice once with `httpd` as the package name, and once with `memcached` as the package name.

We can also iterate through hashes and reference subkeys.

```
- name: Add user and assign groups
  user:
    name: "{{ item.name }}"
    group: "{{ item.groups }}"
    state: present
  with_items:
    - name: user1
      groups: wheel
    - name: user2
      groups: root
```

With items can also have a non-yaml hash

```
- name: Add user and assign groups
  user:
    name: "{{ item.name }}"
    group: "{{ item.groups }}"
    state: present
  with_items:
    - { name: user1, groups: wheel }
    - { name: user2, groups: root }
```

Nested Loops

Loops can also be nested:

```
vars:
  ip_addresses:
    - 10.10.20.21
    - 10.10.20.22
    - 10.10.20.23
  ports: ['80','443']
tasks:
  - name: give multiple users access to multiple databases
    mysql_user:
      name: "{{ item[0] }}"
      priv: "{{ item[1] }}.*:ALL"
      append_privs: yes
      password: "password"
    with_nested:
      - [ '', 'user2' ]
      - [ 'database1', 'database2', 'database3' ]
```

You can also use a variable with a list.

```
vars:
  users:
    - user1
    - user2
  databases:
    - database1
    - database2
    - database3
tasks:
  - name: give multiple users access to multiple databases
    mysql_user:
      name: "{{ item[0] }}"
      priv: "{{ item[1] }}.*:ALL"
      append_privs: yes
      password: "password"
    with_nested:
      - {{ users }}
      - {{ databases }}
```

This could help you reuse the same user list and database list for other tasks too.

Looping with Hashes

Lets use a hash of a few datapoints.

```
vars:
  pools:
    pool1:
      servers:
        - { "ip": { "addr": "10.90.130.13", "type": "V4" }}
        - { "ip": { "addr": "10.90.130.15", "type": "V4" }}
    pool2:
      servers:
        - { "ip": { "addr": "10.90.132.13", "type": "V4" }}
        - { "ip": { "addr": "10.90.132.15", "type": "V4" }}
tasks:
  - name: Create the pools
    avi_pool:
      controller: 10.10.27.90
      username: admin
      password: AviNetworks123!
      tenant: admin
      name: "{{ item.key }}"
      state: present
      enabled: false
      health_monitor_refs:
        - '/api/healthmonitor?name=System-HTTP'
      servers: "{{ item.value.servers }}"
    with_dict:
      "{{ pools }}"
```

This would iterate through our list of pools, and with their different servers create 2 new pools via the `avi_pool` module.

Looping with Files

We can also loop over files for example printing content of a file we have locally. Files are either absolute location or relative.

```
- tasks:
    - debug: msg={{ item }}
      with_file:
        - file01
        - file02
```

This will print out both file01 and file02 to our screen.

Looping with Fileglobs

Using a fileglob we can output all files in a directory that match a pattern, non-recursively.

```
- tasks:
    - name: ensure remote directory exists
      file: dest="/opt/mydirectory" state=directory

    - name: deploy my files
      copy: src="{{ item }}" dest="/opt/mydirectory/"
      with_fileglob:
        - "mydirectory/*"
```

Note: When using a relative path with `with_fileglob` in a role, Ansible resolves the path relative to the roles/<rolename>/files directory.

Looping with Subelements

Lets take an example where we need to setup a set of servers before deploying Avi Controllers to them. We need to create users, and allow their SSH keys to be used. We will define a set a vars and then use those to create our users with associated keys.

```
users:
  - name: user1
    ssh_pub_key:
      - /tmp/user1/ssh/user1.pub
      - /tmp/user1/ssh/otherkey.pub
  - name: user2
    ssh_pub_key:
      - /tmp/user2/ssh/user2.pub
```

```
- name: Create User
  user:
    name: "{{ item.name }}"
    state: present
    generate_ssh_key: yes
  with_items:
    - "{{ users }}"
```

(continues on next page)

(continued from previous page)

```
- name: Set authorized ssh key
  authorized_key:
    user: "{{ item.0.name }}"
    key: "{{ lookup('file', item.1) }}"
  with_subelements:
     - "{{ users }}"
     - ssh_pub_key
```

Registered Variables with Loops

When using `register` with a loop the results will contain a list of the responses. This can be very useful when registering host creation and getting the servers from the array from the task.

```
tasks:
  - name: Provision a set of instances
    ec2:
      aws_access_key: "{{ec2_access_key}}"
      aws_secret_key: "{{ec2_secret_key}}"
      key_name: aws_key
      group: server
      instance_type: t2.micro
      image: ami-123456
      wait: true
      exact_count: 1
      count_tag:
        Name: "{{ item }}"
      instance_tags:
        Name: "{{ item }}"
    register: ec2
    with_items:
      - server1
      - server2
      - server3
```

This would then return an array of results, which each would contain IP addresses and other data from the AWS api call. We can then use the returned data to add servers to a group, create an array to submit to a controller as new pool servers as well.

```
# Value returned from Ansible
```

Looping over Inventory

If you want to loop over a group of inventory hosts or a subset you can use `with_items` and the variable `play_hosts` or using the `group` variable. Play hosts would use the hosts assigned to the current play in the playbook. Group variable usage is `groups['groupname']` and would use all of the hosts in the declared group.

This example would give us a list of all the hosts in the current play.

```
- debug: msg={{ item }}
  with_items:
    - "{{ play_hosts }}"
```

This example would give us a list of all the hosts in the group we specified in this case `servers`.

```
- debug: msg={{ item }}
  with_items:
    - "{{ groups['servers']}}"
```

2.2.11 Blocks

As long as you are using Ansible 2.0 and later, you can use blocks. These help in logically grouping a set of tasks in your playbooks, as well as error handling. Most options you can pass for a task will work for a block. However at this moment `with_items`, and other loops do not work.

Grouping tasks

```
tasks:
  - block:
      - yum: name=httpd state=present

      - template: src=httpd.j2 dest=/etc/httpd/conf/httpd.conf

      - service: name=httpd state=started enabled=true

    when: ansible_distribution == "CentOS"
    become: true
    become_user: root
```

This is an example of creating a block of tasks which are only to run when the OS Distribution is CentOS. It helps clean up cases where you may have multiple tasks that rely on the same `when` statements, or other commonalities that can be combined.

Error Handing

Another nice feature of blocks includes the ability to add error handling to your playbooks. You can recover or create a process to roll back your tasks if a failure occurs. This can have great benefits for some use cases.

There are a few options in block we can use such as of course `block`, but also `rescue`, and `always`.

```
tasks:
  - block:
      - debug: msg="Executing task 1"
      - shell: /bin/false
      - debug: msg="Executing task 2"
    rescue:
      - debug: msg="The previous command failed"
      - debug: msg="Execute rescue task 1"
      - shell: /bin/true
    always:
      - debug: msg="I will always work regardless of any failures prior"
```

To explain what happens here:

First we start executing tasks in the block. However the second task `shell` returns `false` which will fail the task. This results in the skip of "task 2". Then the block will load the `rescue` portion of the block. We can use this section to both inform the user that a block failed, and issue commands to fix it. Then the block will progress. It will always run the section `always` as the `always` section will run regardless of error condition on the `block` or `rescue` sections. Which could be handy to proceed or change data for each run successful or not.

2.2.12 Best Practices

There are a few key things we need to keep in mind when dealing with the deployment of Avi with Ansible.

- A change in Ansible role values will result in a change in the Avi deployment. Which will likely result in a controller being taken offline for a bit as it restarts with the new service configuration.
- Always, Always verify your values to desired values before execution. Ansible is powerful, it will do what you tell it. Remember: Garbage in, Garbage out.
- Automation makes it less likely to make a mistake, but automation also allows you scale the mistake very rapidly. Always test, or have someone else verify your code prior to execution.

Repository Layout

We recommend following the structure described by Ansible here http://docs.ansible.com/ansible/playbooks_best_practices.html#directory-layout

Basic

```
production                # production inventory file
stage                     # stage inventory file

group_vars/
  group1                  # variables assigned to hosts in group1
  group2                  # variables assigned to hosts in group2
host_vars/
  hostname1               # variables assigned to 'hostname1'
  hostname2               # variables assigned to 'hostname2'

library/                  # custom modules go here

avicontroller.yml         # playbook that can deploy your controller or
                          # build your controller

avise.yml                 # playbook that can prepare your SE servers
                          # for the Linux Server Cluster Avi deploy

create_vip.yml            # playbook that uses modules to create vip
...                       # if you have too many of these,
                          # think about combining tasks into roles,
                          # and providing variables to roles
```

(continues on next page)

(continued from previous page)

```
roles/
  avinetworks.avisdk             # avisdk role downloaded from Galaxy
  avinetworks.avicontroller      # avicontroller role downloaded from Galaxy
  avinetworks.docker             # docker role downloaded from Galaxy
  automation_role/               # this can be the custom role you create
    tasks/
      main.yml                   # tasks that are common to your playbooks
    vars/
      main.yml                   # vars you set for your tasks within the role
    defaults/
      main.yml                   # default values you that can provide
                                 # defaults to your role tasks
                                 # and can be overriden by playbook
    library/                     # can also contain custom modules
```

Alternative

```
production/
  hosts                          # production inventory file
  group-vars/
    group1                       # variables assigned to hosts in group1
    group2                       # variables assigned to hosts in group2
  host_vars/
    hostname1                    # variables assigned to 'hostname1'
    hostname2                    # variables assigned to 'hostname2'
stage/
  hosts                          # stage inventory file
  group-vars/
    group1                       # variables assigned to hosts in group1
    group2                       # variables assigned to hosts in group2
  host_vars/
    hostname1                    # variables assigned to 'hostname1'
    hostname2                    # variables assigned to 'hostname2'

library/                         # custom modules go here

avicontroller.yml                # playbook that can deploy your controller or
                                 # build your controller

avise.yml                        # playbook that can prepare your SE servers
                                 # for the Linux Server Cluster Avi deploy

create_vip.yml                   # playbook that uses modules to create vip
...                              # if you have too many of these,
                                 # think about combining tasks into roles,
                                 # and providing variables to roles


roles/
  avinetworks.avisdk             # avisdk role downloaded from Galaxy
  avinetworks.avicontroller      # avicontroller role downloaded from Galaxy
  avinetworks.docker             # docker role downloaded from Galaxy
  automation_role/               # this can be the custom role you create
    tasks/
```

(continues on next page)

(continued from previous page)

```
    main.yml                    # tasks that are common to your playbooks
  vars/
    main.yml                    # vars you set for your tasks within the role
  defaults/
    main.yml                    # default values you that can provide
                                # defaults to your role tasks
                                # and can be overriden by playbook
  library/                      # can also contain custom modules
```

Use Dynamic Inventory when Possible

If you are deploying Avi in a cloud environment using Ansible, it's best to use Dynamic Inventory. Dynamic Inventory allows an inventory script to be executed and based on parameters return specific hosts based on tags or other values. For further information please see: http://docs.ansible.com/ansible/intro_dynamic_inventory.html

How to seperate Staging vs Production

When using a static inventory, you will want to seperate staging vs production. These same practices can be applied to Dynamic Inventory as well. For example, using a AWS Tag "environment:production" would group systems in the *ec2_tag_environment_production* group. Our recommendation is to seperate your static hosts between two files for staging and production. This will prevent any possible confusion between what hosts are being executed on prior to running a playbook. An example run would look like

```
ansible-playbook -i production myplaybook.yml
```

Running it this way will ensure that only the production hosts are being executed against.

Version Control

The use of Version Control software is extremely important. It will help maintain an audit trail, and allow others to verify code changes prior to pulling them into the master or branch used to execute. It's extremely important to have someone verify configuration changes. A simple typo can easily unintentionally down a service or cause interruption.

Vault

We recommend encrypting anything that includes sensitive information, such as password. Ansible has a feature called Vault, which can by the command `ansible-vault`. Best advice is to create a file named `vars` and `vault`, located in the `group_vars/` directory. In the `vars` file, define all the possible variables needed, including sensitive ones. Then in the `vault` file copy all the sensitive variables over and prefix with `vault_`. Then in the `vars` file point to the matching `vault_` variables. Then using `ansible-vault encrypt vault.yml` to encrypt your sensitive variables. To decrypt on execution use `--ask-vault-pass`. When executing your playbook it will prompt for the decrpytion password.

Ansible's configuration language is put into a Playbook. Playbooks have many components including hosts, variables, tasks, etc. We will cover all of these components and more throughout this section. Another way to look at playbooks is as a instruction for Ansible to orchestrate, provision, deploy, and configure your environment.

Playbooks can contain a list one or multiple plays. Each play can target the same or different systems, but plays are usually targetted to a specific group of systems.

Here is an example playbook containing a single play.

```
---
- hosts: all
  vars:
    controller_version: latest
  roles:
    - role: avinetworks.docker
    - role: avinetworks.avicontroller
      con_version: "{{ controller_version }}"
```

As you can see it's also in YAML format which we've covered earlier.