ANSIBLE

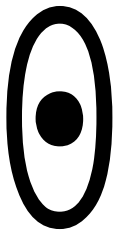
Table of Contents

Typographic Conventions

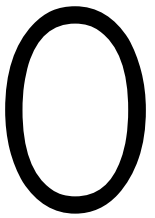
The fonts, layout, and typographic conventions of this book have been carefully chosen to increase readability. Please take a moment to familiarize yourself with them.

A Warning and Solution

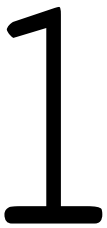
A common problem with computer training and reference materials is the confusion of the numbers "*zero*" and "*one*" with the letters "*oh*" and "*ell*". To avoid this confusion, this book uses a fixed-width font that makes each letter and number distinct.



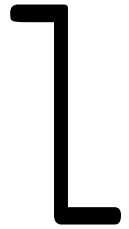
The number "*zero*".



The letter "*oh*".



The number "*one*".



The letter "*ell*".

Typefaces Used and Their Meanings

The following typeface conventions have been followed in this book:

`fixed-width normal` ⇒ Used to denote file names and directories. For example, the `/etc/passwd` file or `/etc/sysconfig/`directory. Also used for computer text, particularily command line output.

`fixed-width italic` ⇒ Indicates that a substitution is required. For example, the string `station`*`X`* is commonly used to indicate that the student is expected to replace *`X`* with his or her own station number, such as `station3`.

`fixed-width bold` ⇒ Used to set apart commands. For example, the **`sed`** command. Also used to indicate input a user might type on the command line. For example, **`ssh -X station3`**.

`fixed-width bold italic` ⇒ Used when a substitution is required within a command or user input. For example, **`ssh -X station`*`X`***.

`fixed-width underlined` ⇒ Used to denote URLs. For example, `http://www.gurulabs.com/`.

variable-width bold ⇒ Used within labs to indicate a required student action that is not typed on the command line.

Occasional variations from these conventions occur to increase clarity. This is most apparent in the labs where bold text is only used to indicate commands the student must enter or actions the student must perform.

Typographic Conventions

Terms and Definitions

The following format is used to introduce and define a series of terms:

deprecate ⇒ To indicate that something is considered obsolete, with the intent of future removal.
frob ⇒ To manipulate or adjust, typically for fun, as opposed to tweak.
grok ⇒ To understand. Connotes intimate and exhaustive knowledge.
hork ⇒ To break, generally beyond hope of repair.
hosed ⇒ A metaphor referring to a Cray that crashed after the disconnection of coolant hoses. Upon correction, users were assured the system was rehosed.
mung (or munge) ⇒ Mash Until No Good: to modify a file, often irreversibly.
troll ⇒ To bait, or provoke, an argument, often targeted towards the newbie. Also used to refer to a person that regularly trolls.
twiddle ⇒ To make small, often aimless, changes. Similar to frob.

When discussing a command, this same format is also used to show and describe a list of common or important command options. For example, the following **ssh** options:

-X ⇒ Enables X11 forwarding. In older versions of OpenSSH that do not include **-Y**, this enables trusted X11 forwarding. In newer versions of OpenSSH, this enables a more secure, limited type of forwarding.
-Y ⇒ Enables trusted X11 forwarding. Although less secure, trusted forwarding may be required for compatibility with certain programs.

Representing Keyboard Keystrokes

When it is necessary to press a series of keys, the series of keystrokes will be represented without a space between each key. For example, the following means to press the "j" key three times: j j j

When it is necessary to press keys at the same time, the combination will be represented with a plus between each key. For example, the following means to press the "ctrl," "alt," and "backspace" keys at the same time: Ctrl + Alt + Backspace. Uppercase letters are treated the same: Shift + A

Line Wrapping

Occasionally content that should be on a single line, such as command line input or URLs, must be broken across multiple lines in order to fit on the page. When this is the case, a special symbol is used to indicate to the reader what has happened. When copying the content, the line breaks should not be included. For example, the following hypothetical PAM configuration should only take two actual lines:

```
password required /lib/security/pam_cracklib.so retry=3➥
       type= minlen=12 dcredit=2 ucredit=2 lcredit=0 ocredit=2
password required /lib/security/pam_unix.so use_authtok
```

Representing File Edits

File edits are represented using a consistent layout similar to the unified **diff** format. When a line should be added, it is shown in bold with a plus sign to the left. When a line should be deleted, it is shown struck out with a minus sign to the left. When a line should be modified, it is shown twice. The old version of the line is shown struck out with a minus sign to the left. The new version of the line is shown below the old version, bold and with a plus sign to the left. Unmodified lines are often included to provide context for the edit. For example, the following describes modification of an existing line and addition of a new line to the OpenSSH server configuration file:

File: /etc/ssh/sshd_config

```
   #LoginGraceTime 2m
-  #PermitRootLogin yes
+  PermitRootLogin no
+  AllowUsers sjansen
   #StrictModes yes
```

Note that the standard file edit representation may not be used when it is important that the edit be performed using a specific editor or method. In these rare cases, the editor specific actions will be given instead.

Lab Conventions

Lab Task Headers

Every lab task begins with three standard informational headers: "Objectives," "Requirements," and "Relevance". Some tasks also include a "Notices" section. Each section has a distinct purpose.

Objectives ⇒ An outline of what will be accomplished in the lab task.
Requirements ⇒ A list of requirements for the task. For example, whether it must be performed in the graphical environment, or whether multiple computers are needed for the lab task.
Relevance ⇒ A brief example of how concepts presented in the lab task might be applied in the real world.
Notices ⇒ Special information or warnings **needed** to successfully complete the lab task. For example, unusual prerequisites or common sources of difficulty.

Command Prompts

Though different shells, and distributions, have different prompt characters, examples will use a `$` prompt for commands to be run as a normal user (like `guru` or `visitor`), and commands with a `#` prompt should be run as the `root` user. For example:

```
$ whoami
guru
$ su -
Password: password
# whoami
root
```

Occasionally the prompt will contain additional information. For example, when portions of a lab task should be performed on two different stations (always of the same distribution), the prompt will be expanded to:

```
stationX$ whoami
guru
stationX$ ssh root@stationY
root@stationY's password: password
stationY# whoami
root
```

Variable Data Substitutions

In some lab tasks, students are required to replace portions of commands with variable data. Variable substitution are represented using italic fonts. For example, *X* and *Y*.

Substitutions are used most often in lab tasks requiring more than one computer. For example, if a student on `station4` were working with a student on `station2`, the lab task would refer to `station`*X* and `station`*Y*

```
stationX$ ssh root@stationY
```

and each would be responsible for interpreting the *X* and *Y* as `4` and `2`.

```
station4$ ssh root@station2
```

Truncated Command Examples

Command output is occasionally omitted or truncated in examples. There are two type of omissions: complete or partial.

Sometimes the existence of a command's output, and not its content, is all that matters. Other times, a command's output is too variable to reliably represent. In both cases, when a command should produce output, but an example of that output is not provided, the following format is used:

```
$ cat /etc/passwd
. . . output omitted . . .
```

In general, at least a partial output example is included after commands. When example output has been trimmed to include only certain lines, the following format is used:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
. . . snip . . .
clints:x:500:500:Clint Savage:/home/clints:/bin/zsh
. . . snip . . .
```

Lab Conventions

Distribution Specific Information

This courseware is designed to support multiple Linux distributions. When there are differences between supported distributions, each version is labeled with the appropriate base strings:

R ⇒ Red Hat Enterprise Linux (RHEL)
S ⇒ SUSE Linux Enterprise Server (SLES)
U ⇒ Ubuntu

The specific supported version is appended to the base distribution strings, so for Red Hat Enterprise Linux version 6 the complete string is: R6.

Certain lab tasks are designed to be completed on only a sub-set of the supported Linux distributions. If the distribution you are using is not shown in the list of supported distributions for the lab task, then you should skip that task.

Certain lab steps are only to be performed on a sub-set of the supported Linux distributions. In this case, the step will start with a standardized string that indicates which distributions the step should be performed on. When completing lab tasks, skip any steps that do not list your chosen distribution. For example:

1) [R4] *This step should only be performed on RHEL4.*
Because of a bug in RHEL4's Japanese fonts...

Sometimes commands or command output is distribution specific. In these cases, the matching distribution string will be shown to the left of the command or output. For example:

```
     $ grep -i linux /etc/*-release | cut -d: -f2
[R6] Red Hat Enterprise Linux Server release 6.0 (Santiago)
[S11] SUSE Linux Enterprise Server 11 (i586)
```

Action Lists

Some lab steps consist of a list of conceptually related actions. A description of each action and its effect is shown to the right or under the action. Alternating actions are shaded to aid readability. For example, the following action list describes one possible way to launch and use **xkill** to kill a graphical application:

Alt+F2	Open the "Run Application" dialog.
xkill Enter	Launch xkill. The cursor should change, usually to a skull and crossbones.
Click on a window of the application to kill.	Indicate which process to kill by clicking on it. All of the application's windows should disappear.

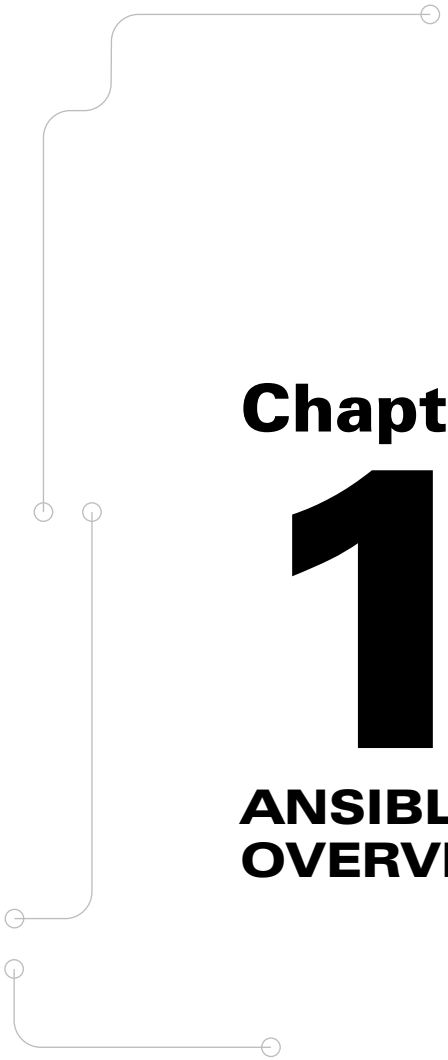
Callouts

Occasionally lab steps will feature a shaded line that extends to a note in the right margin. This note, referred to as a "callout," is used to provide additional commentary. This commentary is never necessary to complete the lab succesfully and could in theory be ignored. However, callouts do provide valuable information such as insight into why a particular command or option is being used, the meaning of less obvious command output, and tips or tricks such as alternate ways of accomplishing the task at hand.

```
[S10] $ sux -
      Password: password
      # xclock
```

- On SLES10, the sux command copies the MIT-MAGIC-COOKIE-1 so that graphical applications can be run after switching to another user account. The SLES10 su command did not do this.

Content



Chapter 1

ANSIBLE OVERVIEW

PARA INSTALAR ANSIBLE EN LAS INSTANCIAS EC2
sudo yum update -y
sudo yum install -y python3 nmap git curl
python3 -m pip install --upgrade pip
python3 -m pip install boto3
python3 -m pip intall ansible

NOTA: ESTO INSTALA ANSIBLE SOBRE PYTHON3
SI SE INSTALA DESDE REPOSITORIO, ANSIBLE USA PYTHON2 QUE EN AWS FALLA PARA DOCKER

Why Ansible?

Modern computing infrastructure:
- complex
- massive scope/scale
- demanding high reliability

Ansible provides tooling to:
- Provision
- Configure
- Orchestrate

Large set of core modules
- Easy to create you own custom modules in Python

Huge community sharing reusable roles

Realities of Modern Computing Infrastructure

Like it or not, the modern world is built upon a complex, expansive, computing infrastructure. Furthermore, the reliability requirements placed upon it necessitate the ability to react quickly to outages, and ever changing business needs. For anything but the simplest of deployments, these requirements quickly exceed the abilities of even experienced IT admins. Tooling that can automate, and bring consistency to the process becomes a necessity.

Traditionally, sysadmins tended to either create their own solutions, or rely on institutional policies and procedures passed down by their predecessors. While custom tooling will always have value, most organizations have seen the value in adopting industry proven solutions. Ansible is somewhat unique in this arena because it tackles the problems of provisioning, configuration management, and orchestration, whereas many products that would purport to compete with it are focused almost exclusively on solving the configuration management problem.

Provision Infrastructure

Ansible ships with many core modules that provide methods of interfacing with computing infrastructure. These modules read definitions stored in simple YAML files and can then provision their respective computing components. For example, an Ansible playbook could be used to:

1. Connect to a router and switch to define a new VLAN and connect it to some number of ports.
2. Bootstrap a new physical machine connected to that LAN, installing a base OS.
3. Configure the base OS on the physical machine to act as a hypervisor.
4. Interface with the VM hypervisor and create new VMs (with all their associated resources.)
5. Configure the VMs with a container engine (such as Docker) so they can host containers.
6. Provision some number of containers on the host providing some service.

Configuration Management

Once infrastructure is provisioned, it still must be configured. Furthermore, configuration is not just a one-time event. Changes introduced by admins, the roll-out of a new application version, or just simple security patching, can all require changes to configuration over time. Ansible provides a powerful declarative method of specifying the desired configuration state for your infrastructure. Running a playbook can quickly bring a set of resources back to a good configuration state. Ansible requires very little of the systems it manages and comes with a comprehensive set of modules to tackle common tasks with an extensible framework to use custom modules if needed.

It is important to note that although Ansible can fix configuration drift, it does not monitor for it. In other words, resources not explicitly controlled by tasks and playbooks can change without Ansible noticing. Using tools such as file integrity checkers, or large scale use

of source control systems for configs can both help solve this problem.

Orchestrate IT Tasks

Once infrastructure is provisioned, and configured, there are still orchestration problems that must be solved:

- How can I safely restart this complex, interdependent stack of services spread across many systems?
- How do automate adding my new web server into an existing load-balanced pool?
- How can I build a set of test machines and perform a series of functional tests every time a new version of my app is pushed to my source control system?

Ansible leverages its provisioning and configuration management abilities to tackle these larger scale orchestration needs. For example, in the case of applying security patches to a pool of web servers Ansible could:

1. Deploy the new web server that has the new security patches in a test environment.
2. Verify it passes a variety of health checks.
3. Tear down the test instance.
4. Remove one production web instance from the load balancer's pool.
5. Apply the security patch to the production web instance.
6. Verify it passes health tests, and if so, add it back into the load balancer's pool.
7. Repeat the last three steps (with the desired level of parallelism) until the entire web pool is patched.

"Not my problem." --Ansible

As explained, Ansible has powerful applications in provisioning, configuration management, and orchestration. Two other important areas where tooling is commonly used are: monitoring & alerting, and logging. While Ansible can certainly be used to deploy and manage the services needed to handle these tasks, it doesn't try to solve them itself.

Large Set of Core Modules, and Community Sharing Reusable Roles

Ansible ships with a large set of core modules that cover many common system provisioning, configuration, and orchestration tasks. Modules exist not only for common Linux administration tasks, but also for many common hardware infrastructure devices and cloud providers. If the shipped library of modules does not cover your specific use case, custom modules can easily be written in Python to extend Ansible abilities. Ansible index of the core modules can be found at: http://docs.ansible.com/ansible/latest/modules_by_category.html.

A large community also exists that collaborates and shares reusable roles. The central hub for sharing roles is Ansible Galaxy: https://galaxy.ansible.com/.

ANSIBLE LO MAS IMPORTANTE ES QUE PERMITE LA IDEMPOTENCIA DE SU CÓDIGO:
SIEMPRE QUE EJECUTES LA MISMA VERSIÓN DE UN PLAYBOOK SIN MODIFICAR EL RESULTADO FINAL ES EL MISMO.

Overview of Architecture

Simple & Secure
Control machine requirements
- Python 2.7, or Python 3.5+
- Windows not supported

Managed node requirements
- Python 2.6+, or Python 3.5+
- SSH (ideally SFTP) or alternate supported transport
- Additional Python libraries (depending on Ansible Modules used)

Inventory, Roles, Modules, Tasks, Playbooks, Transports

Simple, Secure, Easy to Use

Ansible is designed to have a minimum of moving parts and to be quick and easy to deploy. It is client-server in design, but unlike most of its competitors, it is also agent-less. The core Ansible commands can be installed onto a single host which then acts as a controller (or control node). This controller commonly uses SSH as a transport to connect to other machines. Python modules, and other needed files are pushed across the transport connection to a temporary directory and executed on the managed system. Ansible is written in Python and requires core Python to be present on both the controller and managed systems.

The default agentless use of Ansible is a push architecture, with commands executed on the control machine resulting in tasks being executed on the managed systems. Ansible can also be configured in a pull architecture by running the `ansible-pull` agent on the managed machines and having them periodically check in with a server, pull the playbooks down locally and execute them.

Important Terms

The following terms are important to understanding Ansible operation:

host inventory ⇒ A file that lists the systems to be managed. Often used to define groups of systems. May define variables, and behavioural config properties. Default file is `/etc/ansible/hosts`.

module ⇒ Python code pushed to the managed system, executed, and do the actual work. Also sometimes called task plugins, or library plugins.

task ⇒ Written in YAML, and contain a reference to module along with a set of configuration properties, and perhaps conditions. They are the basic unit of work for Ansible.

playbook ⇒ A YAML file with a collection of tasks which describe a desired system state. Often also contain variables, conditionals, and loops.

role ⇒ A self contained, reusable collection of: playbooks, templates, variables, etc. A collection of one or more roles can then be assigned to systems listed in the inventory.

transport ⇒ The connection method used between the control node and the managed system. Most commonly OpenSSH with a fall-back to Paramiko (Python based SSHv2 protocol implementation). Can also be direct local execution, execution in a container namespace (think **`docker exec`**), and other more esoteric methods like chroot and jail.

From the Ansible documentation: "If Ansible modules are the tools in your workshop, playbooks are your instruction manuals, and your inventory of hosts are your raw material."

QUIZ: Architecture

Quiz and group discussion

Quiz: Ansible Architecture

1. Which of the following areas of automation is Ansible used for?
   - Logging
   - Provisioning
   - Orchestration
   - Configuration management
   - Monitoring, and Alerting
2. What programming language is Ansible written in?
   - Ruby
   - Python
   - Javascript (NodeJS)
   - Perl
3. An Ansible operational playbook could be used handle the graceful update of a pool of web servers behind a load-balancer.
   - True
   - False
4. Which of the following is the correct name for an Ansible file containing a collection of tasks that can be run to bring a system into a specified state?
   - cookbook
   - module
   - playbook
   - manifest
   - states
5. What is the most common transport used to pass data between the controller and managed systems?
   - vxlan tunnels
   - SSH
   - UNIX sockets
   - UDP Multicast
   - HTTPS
6. What type of structural data file format is used when writing tasks?

Inventory

Systems to be managed
- /etc/ansible/hosts (default)
- INI-like format (default) -- YAML or other based on inventory plugins
- hosts and groups

Variables

Behavioral inventory parameters

Inventory

An inventory file defines what systems Ansible will manage. Inventory files can be very simple consisting of nothing but a list of resolvable hostnames and/or IP addresses (listed one per line). By default, Ansible commands use the `/etc/ansible/hosts` file as the inventory, but another file can be specified via the config, or manually when invoking the Ansible command.

Inventory Groups

If many hosts will need the same playbooks run against them, then a group can be defined in the inventory file. Group names are listed in square brackets with the list of contained hosts following. To create groups of groups, use the special `:children` suffix when defining the group name, then list the included groups.

Note that hosts can be listed in more than one group. For example, groups might be defined that organize systems by there role, and additional groups that organized them by their availability zone, region, data center, etc. As an example, consider the following simple inventory file:

File: hosts FICHERO hosts1 EN EL DIRECTORIO DE LABS

```
lb.example.com     # single host

[web]              # group of 3 hosts
web1.example.com
web2.example.com
192.0.2.42         # can use IP instead

[db]
db1.example.com
db2.example.com

[zone1]            # hosts can be in multiple groups
web1.example.com
web2.example.com
db1.example.com

[zone2]
web3.example.com
db2.example.com

[app1:children]    # a group of groups
db
web
```

Using Ranges in Host Inventory

If multiple system names differ only by a simple iterator, then you can match the entire range with a single entry. For example, `db[001:100].example.com` would expand to the match 100 systems with the lower numbered ones having their names zero padded. For the same list without zero padding, use: `db[1:100].example.com`.

Defining Variables in Inventory File

Ansible's use of variables allow its playbooks to avoid hard coding system specific details letting them adapt to a wide variety of systems and conditions. Variables can be defined in many places, and a more complete coverage of best practices is found later in the course. Variables defined in the inventory file can apply to hosts, or groups, and will be overridden by variables defined via any of the other methods (other than role defaults). A best practice is to only include variables specific to connecting to the host or group. The following example shows defining both host and group variables:

File: hosts

```
. . . snip . . .
localhost       ansible_connection=local
jumpnode        ansible_host=10.100.0.250 ansible_port=2222

[web:vars]
web_port=8000
ansible_user=web
ansible_become_user=webadmin
ansible_ssh_private_key_file=files/web_rsa
```

Defining Variables via Directories

To keep the inventory file clean and focused on its primary purpose, variables can be moved in a directory structure instead. By default Ansible will read files found under the `/etc/ansible/host_vars`, and `/etc/ansible/group_vars` directories. The files should be named after the hosts or groups they correspond to, and should be structured as YAML or JSON files. For example:

File: /etc/ansible/host_vars/jumpnode.yaml

```
---
ansible_host=10.100.0.250
ansible_port=2222
```

File: /etc/ansible/group_vars/web.yaml

```
---
ansible_user=web
ansible_become_user=webadmin
ansible_ssh_private_key_file=files/web_rsa
```

Variable Types

In general, variables simply associate some data with a name. Playbooks, and any templates they reference, can then refer to the variables, and make use of the data they contain. While the examples shown thus far have been simple key/value pairs, Ansible supports creating variables containing more complex data structures such as hashes and arrays. A handful of variable names (usually beginning with `ansible_` are reserved and used to control internal Ansible behaviors. See http://docs.ansible.com/ansible/latest/intro_inventory.html#list-of-behavioral-inventory-parameters for a full list of these behavioral parameters.

Inventory Patterns

Select matching hosts from inventory

- `all`, or `*`
- *pat1*`:`*pat2* → pattern 1 OR pattern 2
- *pat1*`:!`*pat2* → pattern 1 AND NOT pattern 2
- *pat1*`:&`*pat2* → pattern 1 AND pattern 2
- `~`*pat1* → pattern 1 (as REGEX)
- *pat1*`[subscript:`*X*`]` → slice of group

Host Inventory Patterns

When running playbooks, and ad hoc commands, you designate which hosts to operate on. If they are well designed, the groups defined in the inventory file will often be sufficient. However, Ansible provides a rich set of operators that allow groups to be logically combined and subsets of groups to be selected. The `ansible --list-hosts` option causes it to just return the list of systems matching the specified patterns. The following examples show the use of these pattern matching methods applied to this simple inventory file:

File: `hosts` FICHERO hosts2 EN EL DIRECTORIO

```
lb.example.com

[web]
web[1:3].example.com
[db]
db[1:2].example.com
[zone1]
web[1:2].example.com
db1.example.com
[zone2]
web3.example.com
db2.example.com
```

All hosts pattern

```
$ ansible --list-hosts all
  hosts (6):
    lb.example.com
    web1.example.com
    web2.example.com
    web3.example.com
    db1.example.com
    db2.example.com
```

PROBAR ESTOS COMANDO CON EL FICHERO hosts2 DEL DIRECTORIO

: - Logical OR

Separate patterns with a colon to match systems matching any of the patterns. For example:

```
$ ansible --list-hosts lb.example.com:db
  hosts (3):
    lb.example.com
    db1.example.com
    db2.example.com
```

:! - Logical Exclusion

Select systems matching the first pattern, that don't also match the second pattern. For example:

```
$ ansible --list-hosts 'web:!zone2'
  hosts (2):
    web1.example.com
    web2.example.com
```

:& - Logical Exclusion

Select systems matching both the first and second patterns (the intersection of the groups). For example:

```
$ ansible --list-hosts 'db:&zone1'
  hosts (1):
    db1.example.com
```

~ - Regular Expression

Patterns preceded by a tilde are treated as regular expressions and all matching systems are returned. For example:

```
$ ansible --list-hosts '~(web|db)2'
  hosts (2):
    web2.example.com
    db2.example.com
```

group[index] - Position Within Group, or Slice

Individual elements of a group can be selected by specifying indexes within square brackets. For example:

```
$ ansible --list-hosts 'web[0]'
  hosts (1):
    web1.example.com
$ ansible --list-hosts 'web[-1]'
  hosts (1):
    web3.example.com
$ ansible --list-hosts 'web[1:2]'
  hosts (2):
    web2.example.com
    web3.example.com
$ ansible --list-hosts 'web[1:]'
  hosts (2):
    web2.example.com
    web3.example.com
```

Inventory Plugins

Extends ability to compile inventory
Most plugins are disabled by default
- default enabled: host_list, script, yaml, ini, auto

ansible-inventory
- validate lists
- examine vars
- export data

Enabling Inventory Plugins

Most inventory plugins shipped with Ansible are disabled by default and need to be whitelisted in your `ansible.cfg` file in order to function. The `enable_plugins` config property defines the plugin list and the order of execution. Some inventory plugins rely on external commands and can potentially take a long time to run or place additional load on infrastructure. Give thought to which are needed and only enable those plugins for maximum performance.

A list of the available inventory plugins can be produced as follows:

```
# ansible-doc -t inventory -l
advanced_host_list  Parses a 'host list' with ranges
auto            Loads and executes an inventory plugin
      specified in a YAML config
aws_ec2         ec2 inventory source
aws_rds         rds instance source
azure_rm        Azure Resource Manager inventory plugin
constructed     Uses Jinja2 to construct vars and groups
      based on existing inventory.
foreman         foreman inventory source
gcp_compute     Google Cloud Compute Engine inventory source
generator       Uses Jinja2 to construct hosts and groups
      from patterns
host_list       Parses a 'host list' string
ini             Uses an Ansible INI file as inventory source.
k8s             Kubernetes (K8s) inventory source
nmap            Uses nmap to find hosts to target
openshift       OpenShift inventory source
openstack       OpenStack inventory source
scaleway        Scaleway inventory source
script          Executes an inventory script that returns JSON
tower           Ansible dynamic inventory plugin for Ansible
      Tower.
virtualbox      virtualbox inventory source
vmware_vm_inventory VMware Guest inventory source
vultr           Vultr inventory source
yaml            Uses a specific YAML file as an inventory source.
```

Details for each can be obtained by specifying the desired plugin instead of the list option:

```
# ansible-doc -t inventory advanced_host_list
> ADVANCED_HOST_LIST    (/usr/lib/python2.7/site-packages/ansible
      /plugins/inventory/advanced_host_list.py)

Parses a host list string as a comma separated values of hosts
and  supports host ranges. This plugin only applies to inventory
sources  that are not paths and contain at least one comma.



EXAMPLES:
# simple range
    # ansible -i 'host[1:10],' -m ping

    # still supports w/o ranges also
```

ansible-inventory Command

The **ansible-inventory** command is especially useful when first enabling and testing new inventory plugins or dynamic inventory scripts (covered later). It can produce a graph of the entire inventory, just a selection matching a host pattern, or details for just a single host. It can also export the inventory list in a variety of formats. The following examples demonstrate:

Test the use of the advanced_host_list plugin to generate an inventory from a pattern:

```
# ansible-inventory --graph -i "station[01:10].example.com,"
@all:
  |--@ungrouped:
  |  |--station01.example.com
  |  |--station02.example.com
. . . snip . . .
  |  |--station09.example.com
  |  |--station10.example.com
```

Use the aws_ec2 plugin to create custom groups and populate additional host variables (taken from official documentation):

File: demo.aws_ec2.yaml

```
plugin: aws_ec2
regions:
  - us-east-1
  - us-east-2
keyed_groups:
  # add hosts to tag_Name_value groups for ➥
      each aws_ec2 host's tags.Name variable
  - key: tags.Name
    prefix: tag_Name_
    separator: ""
groups:
  # add hosts to the group development if ➥
      any of the dictionary's keys or values is the word 'devel'
  development: "'devel' in (tags|list)"
compose:
  # set the ansible_host variable to connect ➥
      with the private IP address without changing the hostname
  ansible_host: private_ip_address
```

```
# ansible-inventory -i demo.aws_ec2.yml --graph
@all:
  |--@aws_ec2:
  |  |--ec2-12-345-678-901.compute-1.amazonaws.com
  |  |--ec2-98-765-432-10.compute-1.amazonaws.com
. . . snip . . .
  |--@development:
  |  |--ec2-12-345-678-901.compute-1.amazonaws.com
  |  |--ec2-98-765-432-10.compute-1.amazonaws.com
  |--@tag_Name_ECS_Instance:
  |  |--ec2-98-765-432-10.compute-1.amazonaws.com
  |--@tag_Name_Test_Server:
  |  |--ec2-12-345-678-901.compute-1.amazonaws.com
  |--@ungrouped
```

Escriba el texto aquí

EL FICHERO demo.aws.ec2.yaml DEL DIRECTORIO DE LABS ESTÁ MODIFICADO PARA QUE MUESTRE LOS HOSTS DE LA REGION Y MUESTRE AGRUPADOS LOS HOSTS QUE TENGAN EL MISMO LABEL Group

QUIZ: Inventory and Patterns

Quiz and group discussion

1. Ansible host inventory files are written in which of the following syntactical formats?
   - JSON
   - YAML
   - text (INI-like)
   - HTML
2. Variables can also be defined in the host inventory file.
   - True
   - False
3. Which of the following suffixes is used in a host inventory file after a group name to indicate that it contains other groups?
   - :nested
   - :children
   - :groups
4. How do you define group variables in the Ansible host inventory file?
   - (fill in blank)
5. Which of the following patterns would select the third, forth, and fifth hosts from the `srv` group?
   - `srv[3-5]`
   - `srv[3:5]`
   - `srv[3,4,5]`
   - `srv[2:4]`
6. What systems would be selected by the following pattern: `ad:&region1:!zone2`
   - Systems in both the `ad` and `region1` groups that are not listed in the `zone2` group.
   - All systems in the `ad` group, plus those in the `region1` group that are not listed in the `zone2` group.
   - All systems in the `ad` group that are not in the `region1:zone2` child group.

ESTA DEMO NO ES NECESARIO HACERLA LITERAL EN AWS
EL FICHERO LOCAL ansible.cfg ESTA PREPARADO PARA QUE EL FICHERO DE INVENTARIO SEA EL FICHERO LOCAL hosts.yaml
PARA CREAR UN INVENTARIO EJECUTAR ESTE COMANDO:

ansible-inventory -i demo.aws_ec2.yaml --list --yaml > hosts.yaml

DEMO: Introducing Ansible

Verify Ansible is installed
Build and verify a host inventory
- add behavioral properties as appropriate

Verify SSH operation
Enable and use inventory plugins
Sneak peek of upcoming concepts
- run ad hoc command
- run playbook

Verify Ansible is Installed

ESTA PARTE NO DEBE HACERSE EN EL LAB DE AWS

```
[root@server]# mkdir ~/ansible-demo/; cd $_
[root@server ansible-demo]# rpm -qi ansible
. . . output omitted . . .
# ansible --version
ansible 2.7.5
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Aug  2 2016, 04:20:16) [GCC 4.8.5 20150623 (Red Hat 4.8.5-4)]
# ls -l /usr/bin | grep ansible
. . . output omitted . . .
```

Build and Verify a Host Inventory

Ansible needs an inventory file to manage hosts. Let's build one:

ESTA PARTE NO DEBE HACERSE EN EL LAB DE AWS

```
# ansible all --list-hosts
 [WARNING]: provided hosts list is empty, only localhost is available
. . . snip . . .
# echo $(hostname) > /etc/ansible/hosts
# echo "station[1:END].example.com" >> /etc/ansible/hosts  #END is last station number in classroom
# ansible all --list-hosts
  hosts (5):
    server1.example.com
    station1.example.com
    station2.example.com
```

```
. . . snip . . .
```

Verify SSH Operation

```
[root@server]# ansible server1.example.com -m ping
server1.example.com | UNREACHABLE! => {
    "changed": false,
    "msg": "Failed to connect to the host via ssh: Permission denied (publickey,gssapi-keyex,gssapi-with-mic,password).\r\n",
    "unreachable": true
}
```

PROBAR CONTRA UNO DE LOS HOSTS DE AWS

Ansible SSH key is not installed as trusted on local server fix by adjusting the transport type, then test for normal stations:

```
# sed -i '/server/ s/$/ ansible_connection=local/' /etc/ansible/hosts
# ansible server1.example.com -m ping
server1.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
# ansible station1.example.com -m ping
The authenticity of host 'station1.example.com (10.100.0.1)' can't be established.
ECDSA key fingerprint is bf:31:61:b3:99:10:e1:d8:37:26:d7:23:70:4e:a6:87.
Are you sure you want to continue connecting (yes/no)? Ctrl+C
```

NO ES NECESARIO EL EQUIPO LOCAL YA ESTA CACHEADO SU CERTIFICADO

The host key for the station is not know. There are many ways to fix this for example, the Ansible config could be modified to pass options to SSH causing it to skip the host key check. Best is to just build a proper `.ssh/known_hosts` file. Test disabling host key verification, then populate a proper `authorized_keys` file:

```
# ANSIBLE_HOST_KEY_CHECKING=False ansible -m ping station1.example.com
station1.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
# ssh-keyscan station{1..END}.example.com >> ~/.ssh/known_hosts
. . . output omitted . . .
# ansible all -m ping
station1.example.com | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
. . . snip . . .
```

EL CHEQUEO DE CLAVES PUEDE PROVOCAR ERROR EN UN PLAYBOOK SI ES LA PRIMERA VEZ QUE SE EJECUTA Y NO HEMOS ENTRADO NUNCA LA SOLUCION ES AL MENOS HACER UN SSH A CADA HOST (INCLUIDO EL LOCALHOST) PARA QUE CACHEE LA CLAVE. OTRA SOLUCION ES EL COMANDO SSH-KEYSCAN QUE SE INDICA A LA IZQUIERDA, O USAR LA VARIABLE ANSIBLE_HOST_KEY_CHECKING=FALSE PARA QUE NO SE CHEQUEE

Enable and Use Inventory Plugins

LOS NOMBRES QUE SE USAN PUEDEN SER INVENTADOS

Get a baseline behavior showing the default plugins can't expand range based clauses and instead treat them literally:

```
# ansible-inventory -i 'station[1:6].example.com,' --graph
@all:
  |--@ungrouped:
  |  |--station[1:6].example.com
```

Enable and test the nmap and advance_host_list inventory plugins:

File: /root/ansible-demo/ansible.cfg

```
+ [inventory]
+ enable_plugins = advanced_host_list, yaml, nmap
```

EL FICHERO ansible.cfg DEL DIRECTORIO LOCAL TIENE YA ESTA CONFIGURACION Y OTRAS PARA ELEVAR LOS PRIVILEGIO DE ROOT Y USAR EL INVENTARIO LOCAL

```
# ansible-inventory -i 'station[1:6].example.com,' --graph
@all:
  |--@ungrouped:
  |  |--station1.example.com
  |  |--station2.example.com
. . . snip . . .
```

Test the nmap plugins' ability to generate an inventory based on the machines it can ping:

File: hosts.yaml

```
+ plugin: nmap
+ strict: False
+ address: 10.100.0.0/24
```

ES EL FICHERO demo.hosts_nmap.yaml EN EL DIRECTORIO DE LOS LABS TARDA MUCHO EN RESPONDER YA QUE HACE UN ESCANEO NMAP EN CADA TARGET

```
# yum install -y nmap
. . . output omitted . . .
# ansible-inventory -i hosts.yaml --graph
@all:
  |--@ungrouped:
  |  |--server1.example.com
  |  |--station1.example.com
  |  |--station2.example.com
  |  |--station3.example.com
```

PUEDE TARDAR BASTANTE EN ESCANEAR EL RANGO

Examine the variable data collected by the nmap plugin for a single host. Then construct a static inventory in YAML based on the dynamically collected info::

```
# ansible-inventory -i hosts.yaml --host station1.example.com --yaml
```

PUEDE TARDAR BASTANTE EN ESCANEAR EL RANGO

```
ip: 10.100.0.1
ports:
- port: '22'
  protocol: tcp
  service: ssh
  state: open
- port: '111'
  protocol: tcp
  service: rpcbind
  state: open
# ansible-inventory -i hosts.yaml --list --yaml > hosts2.yaml
# ansible -i hosts2.yaml --list-hosts all
  hosts (7):
    station1.example.com
    station2.example.com
. . . snip . . .
    server1.example.com
# rm -f hosts* ansible.cfg  #cleanup
```

PUEDE TARDAR BASTANTE EN ESCANEAR EL RANGO

Sneak Peek - Ad hoc Commands

Create a group containing the classroom stations, and run a simple command across all those stations:

```
# sed -i '/station/i[stations]' /etc/ansible/hosts
# ansible stations -m command -a "uptime"
station3.example.com | SUCCESS | rc=0 >>
 15:39:55 up  1:47,  2 users,  load average: 0.08, 0.03, 0.05
. . . snip . . .
# ansible stations -m shell -a "w | head -n1 | tee /tmp/load" -o
station4.example.com | SUCCESS | rc=0 | (stdout)  15:46:11 up  1:53,  2 users,  load average: 0.00, 0.01, 0.05
. . . snip . . .
```

COMANDO AD-OC. SE DEBEN PROBAR CONTRA EL GRUPO DE HOSTS REGITRADOS EN EL INVENTARIO

Clean up /labfiles and populate with content needed for this course:

```
# ansible stations -m file -a ""
# ansible -m file -a "path=/labfiles state=absent" stations
station1.example.com | CHANGED => {
  "changed": true,
  "path": "/labfiles",
  "state": "absent"
}
. . . snip . . .
# ansible -m copy -a "src=/root/GL380-course-files/labfiles/ dest=/labfiles" stations
. . . output omitted . . .
```

NO EJECUTAR ESTE COMANDO Y EL SIGUIENTE YA QUE ES PARA COPIAR EL DIRECTORIO DE LABS QUE NOSOTROS LO DESCARGAMO DE GITHUB

Sneak Peek - Playbooks

Escriba el texto aquí

Create and execute a simple playbook (or copy it from the provided `demo-solutions/` directory) that will configure the remote stations to allow Ansible to connect SSH using the `guru` user and then use **sudo** to become `root` when needed:

File: `ssh_as_guru.yaml`

```
+ ---
+ - hosts: stations
+   tasks:
+   - name: Install server1 root key in guru account
+     authorized_key:
+       user: guru
+       state: present
+       key: "{{ lookup('file', '/root/.ssh/id_rsa.pub') }}"

+   - name: Allow guru to sudo without password
+     copy:
+       dest: /etc/sudoers.d/guru
+       content: "guru ALL=(ALL) NOPASSWD: ALL"

+   - name: Test connecting as guru user and using sudo to become root
+     ping:
+     remote_user: guru
+     become: yes
```

ESTE PLAYBOOK SE USARÍA PARA PODER HACER SSH CON UN USUARIO EN OTRO NODO Y DESDE AHI PODER ENTRAR EN ROOT SIN CONTRASEÑA. NO ES NECESARIO USARLO LAS VM YA TIENEN CONFIGURACIÓN DE SSH

lookup con file es un codigo jinja2 que lee el contenido de un fichero y te lo pasa literal

Ejemplo de cómo ejecutar comandos remotos con otro usuario y haciendo sudo a root

```
# ansible-playbook ssh_as_guru.yaml --syntax-check

playbook: ssh_as_guru.yaml
# ansible-playbook ssh_as_guru.yaml
. . . snip . . .
PLAY RECAP *********************************************************************
station1.example.com       : ok=4    changed=2    unreachable=0    failed=0
station4.example.com       : ok=4    changed=2    unreachable=0    failed=0
. . . snip . . .
# ansible-playbook ssh_as_guru.yaml
. . . snip . . .
PLAY RECAP *********************************************************************
station3.example.com       : ok=4    changed=0    unreachable=0    failed=0
station6.example.com       : ok=4    changed=0    unreachable=0    failed=0
. . . snip . . .
```

NO ES NECESARIO HACER ESTE BLOQUE DE COMANDOS

Execute the same playbook with increasing levels of verbosity:

```
# ansible-playbook -v ssh_as_guru.yaml
# ansible-playbook -vv ssh_as_guru.yaml
# ansible-playbook -vvv ssh_as_guru.yaml
```

Content

Chapter 2

DEPLOYING ANSIBLE

Installing

Ansible consist of

- Simple set of binaries and Python modules on manager node
- No daemons to start
- No database, etc.
- No software installed/running on managed machines

Most major Linux distros have packages available
Can clone source from github for latest

Installing Ansible

The architecture of Ansible requires nothing more that a simple set of binaries, plus their supporting Python modules, to be installed on the selected management node. The hardware requirements are minimal, and even a common laptop could be used to manage a large number of systems. Because Ansible does not rely on a database or other external services, it is especially easy to upgrade in isolation. Although most major Linux distributions have packaged it to make installation easy, many organizations choose to simply clone the source straight from Github.

For Red Hat Linux, the officially supported version is available via the Extras channel. The EPEL repo (https://fedoraproject.org/wiki/EPEL) also hosts packages for RHEL compatible systems like CentOS.

Ubuntu builds are available in the following PPA:
https://launchpad.net/~ansible/+archive/ubuntu/ansible

NO HAY QUE HACER NINGUN PLAYBOOK DE ESTA DEMO YA QUE INSTALAMOS ANSIBLE SOLO EN EL NODO DE CONTROL.

DEMO: Installing Ansible

Ensure stations have Internet access
Configure EPEL repo and install using playbook
Clone Github repo
Build and install from source

Ensure Stations have Internet Access

In the classroom network, all Internet access is provided through server1 which must be configured to act as a router and perform NAT for hosts on the eth0 LAN. Create a playbook (or use the provided solution):

File: internet_for_stations.yaml

```
+ ---
+ - hosts: localhost
+   tasks:
+   - name: Enable Internet facing interface
+     nmcli: conn_name=eth1 autoconnect=yes type=ethernet state=present

+   - name: Configure NAT
+     iptables: table=nat chain=POSTROUTING out_interface=eth1 jump=MASQUERADE

+   - name: Enable routing
+     sysctl: name=net.ipv4.ip_forward value=1 sysctl_set=yes state=present
```

```
[root@server]# cd ~/ansible-demo/

[root@server1 ansible-demo]# ansible-playbook internet_for_stations.yaml
PLAY RECAP *********************************************************************
localhost                  : ok=4    changed=3    unreachable=0    failed=0
```

Configure EPEL Repo and Install Using Playbook

File: install_via_rpm.yaml

```
+ ---
+ - hosts: station1.example.com
+   tasks:

+   - name: Enable EPEL repo
+     yum: name=https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm state=present

+   - name: Install Ansible
+     yum: name=ansible state=present
```

```
# ansible-playbook -v install_via_rpm.yaml
. . . snip . . .
station1.example.com       : ok=3    changed=2    unreachable=0    failed=0

# ansible -a "ansible --version" station1.example.com
station1.example.com | CHANGED | rc=0 >>
ansible 2.7.5
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Aug  2 2016, 04:20:16) [GCC 4.8.5 20150623 (Red Hat 4.8.5-4)]
```

Uninstall via ad hoc Commands

To avoid conflicts (and to see what the real build requirements are), we will back out the packages installed by yum. Currently the yum Ansible module does not support the history functions, so we run commands ad hoc instead of using the module:

```
# ansible -a "yum history" station1.example.com
 [WARNING]: Consider using yum module rather than running yum

station1.example.com | SUCCESS | rc=0 >>

Loaded plugins: langpacks
ID     | Login user               | Date and time    | Action(s)      | Altered
-------------------------------------------------------------------------------
     3 | root <root>              | 2019-01-01 17:07 | Install        |   14
     2 | root <root>              | 2019-01-01 17:06 | Install        |    1  <
     1 | System <unset>           | 2017-11-02 14:34 | Install        | 1318 >
history list
```

```
# ansible -a "yum history info 3 warn=False" station1.example.com
. . . output omitted . . .
# ansible -a "yum -y history undo 3" station1.example.com
. . . output omitted . . .
```

Install via Source (Cloned Github Repo)

Installing via source is slightly more complicated, but can provide access to the absolute latest features.

File: install_via_git.yaml

```
+ ---
+ - hosts: station1.example.com
+   tasks:

+   - name: Install needed packages
+     yum: name=git state=present

+   - name: Clone Ansible repo
+     git: repo=https://github.com/ansible/ansible.git dest=/root/ansible update=no
```

```
# ansible-playbook -v install_via_git.yaml
. . . output omitted . . .
```

Examine the cloned source:

```
# ansible -a "ls -l /root/ansible" station1.example.com
station1.example.com | SUCCESS | rc=0 >>
. . . snip . . .
-rw-r--r--.  1 root root 13633 Jan  1 17:21 Makefile
-rw-r--r--.  1 root root  5370 Jan  1 17:21 README.rst
-rw-r--r--.  1 root root   360 Jan  1 17:21 requirements.txt
-rw-r--r--.  1 root root 11028 Jan  1 17:21 setup.py
. . . output omitted . . .
```

Build and Install from Source

File: install_via_git.yaml

```
   ---
   - hosts: station1.example.com
+    vars:
+      install_root: "/root/ansible"
+      packages:
+      - git
+      - gcc
+      - python
+      - python-devel
+      - python-pip
     tasks:

     - name: Install needed packages
+      yum: name=~~git~~{{ packages }} state=present
+
+    - name: Install build deps
+      pip:
+        requirements: "{{ install_root }}/requirements.txt"
+        extra_args: --upgrade

     - name: Clone Ansible repo
→      git: repo=https://github.com/ansible/ansible.git dest=~~root/ansible~~{{ install_root }}
+
+    - name: Build and install Ansible
+      shell: >
+        python setup.py build;
+        python setup.py install
+      args:
+        chdir: "{{ install_root }}"
+        creates: /usr/bin/ansible
+
+    - name: Get version of Ansible installed
+      command: "{{ install_root }}/bin/ansible --version"
+      register: ANSIBLE_VERSION
+      changed_when: false
+    - debug: msg="{{ ANSIBLE_VERSION.stdout_lines }}"
```

```
# ansible-playbook install_via_git.yaml
. . . output omitted . . .
TASK [debug] ********************************************************************************************************
ok: [station1.example.com] => {
    "msg": [
        "ansible 2.8.0.dev0",
        "  config file = None",
        "  configured module search path = [u'/root/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']",
        "  ansible python module location = /usr/lib/python2.7/site-packages/ansible-2.8.0.dev0-py2.7.egg/ansible",
        "  executable location = /root/ansible/bin/ansible",
        "  python version = 2.7.5 (default, Aug  2 2016, 04:20:16) [GCC 4.8.5 20150623 (Red Hat 4.8.5-4)]"
    ]
}
# rm -rf /usr/lib/python2.7/site-packages/ansible-2.8.0.dev0-py2.7.egg
```

Configuration Files

Multi section (INI style) config read by Ansible

- `$ANSIBLE_CONFIG` environment variable
- `./ansible.cfg` directorio actual
- `~/.ansible.cfg` directorio home de usuario
- `/etc/ansible/ansible.cfg` fichero ansible.cfg generico

Ansible Configuration File

When Ansible commands are invoked on the control system, they look for a config to determine how they should run. Ansible looks for configs in the following four places:

1. If the `ANSIBLE_CONFIG` environment variable is set, then it takes precedence and whatever file it points to is read as the config.
2. If the variable is not set, and a file named `ansible.cfg` exists in the current directory then it is used.
3. If neither of the previous conditions exist, and `.ansbile.cfg` exists in the user's home directory then it is used.
4. If none of the previous conditions exist, then the `/etc/ansible/ansible.cfg` file is used if it exists.
5. If none of the above exists, then internal defaults are used.
6. Options passed via the CLI when running Ansible commands always override options set by any of the above.

Regarding which config file location to use, it is largely personal preference (pick a strategy and stick with it.) The `/etc/ansible/ansible.cfg` is good if the primary role of the system is an Ansible controller and it will be used by many administrators. If a playbook or role is dependent on specific config properties, then it should probably include a suitable base config in its directory, and admins can follow a procedure of changing to the needed directory before running commands. If many different users will use Ansible and require different configs, then go with the `~/.ansible.cfg`.

ansible-config Command

Starting with the 2.4 release, the **`ansible-config`** command can be used to view config details.

list ⇒ list all current configs reading `lib/constants.py` and shows env and config file setting names
dump ⇒ Shows the current settings, merges `ansible.cfg` if specified
view ⇒ Displays the current config file

Ansible Config - General (defaults) Section

The complete documentation for config properties is found at: http://docs.ansible.com/ansible/latest/intro_configuration.html. The following show some of the most commonly used options:

`ask_pass` ⇒ controls whether an Ansible playbook should prompt for a password by default (default no). Generally not needed because SSH keys are used to authenticate instead.
`ask_sudo_pass` ⇒ similar to above, but for password when sudo'ing (default no)
`command_warnings` ⇒ warn the user if they appear to be running a command where a core module would be preferred (default true)
`forks` ⇒ How many connections to spawn in parallel. Set based on the CPU and network load the control node can handle (default 5).
`gathering`, and `gathering_subset` ⇒ control what methods are used to gather facts (default all), and when fact gathering is performed (default implicit = every play).

host_key_checking ⇒ SSH host key verification (default true)
inventory ⇒ location of the inventory file, script, or directory that Ansible will use to determine what hosts it can manage (default `/etc/ansible/ansible.cfg`.
log_path ⇒ file to log module execution events to (default not set - logged to syslog).
module_name ⇒ default module to execute for ad hoc commands (default command).
remote_port ⇒ default SSH port on all of your systems, for systems that didn't specify an alternative value in inventory (default 22).
retry_files_enabled, and **retry_files_save_path** ⇒ whether a failed Ansible playbook should create a `.retry` file (default true and current directory).

Ansible Config - Privilege Escalation Section

become ⇒ causes Ansible to use the defined method (usually **su** or **sudo** to escalate privileges before executing task (default false).
become_method ⇒ method of privilege escalation (default **sudo**).
become_user ⇒ the user to become when privilege escalation is invoked (default root).
become_ask_pass ⇒ ask for password on escalation (default false).

Command Line Config Equivalents

The Ansible CLI also supports passing several common config options when invoking individual Ansible commands. A full list of options can be obtained by running: **ansible --help**.

DEMO: Configuration Files

Examine the default config
Modify the logging settings, inventory file
Change the default SSH port and use compression
Sneak peek: handlers

Examine the Default Config

```
[root@server1]# ansible-config view
. . . output omitted . . .
# egrep -v '^(#|$)' /etc/ansible/ansible.cfg
[defaults]
[inventory]
[privilege_escalation]
[paramiko_connection]
[ssh_connection]
[persistent_connection]
[accelerate]
[selinux]
[colors]
[diff]
```

ESTE COMANDO SE PUEDE EJECUTAR PARA VER LA CONFIGURACION QUE TOMA ANSIBLE

Modify Logging Setting

```
# journalctl | grep ansible-
. . . output omitted . . .
# cd /root/ansible-demo/
```

File: ansible.cfg

```
+ [defaults]
+ log_path = /var/log/ansible.log
```

EL FICHERO ansible.cfg YA ESTA CONFIGURADO, SE LE PUEDE AÑADIR ESTA LINEA

```
# touch /var/log/ansible.log
```

```
# ansible-config view
[defaults]
log_path = /var/log/ansible.log
# ansible-config dump --only-changed
DEFAULT_LOG_PATH(/root/ansible-demo/ansible.cfg) = /var/log/ansible.log
# ansible localhost -m ping
localhost | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
# cat /var/log/ansible.log
2019-01-02 11:16:49,403 p=17054 u=root |  localhost | SUCCESS => {
    "changed": false,
        "ping": "pong"
        }
# ansible demohost -m ping
 [WARNING]: No hosts matched, nothing to do

# echo "demohost ansible_host=station1.example.com" > hosts
# echo "inventory=hosts" >> ansible.cfg
# ansible-config dump --only-changed
DEFAULT_HOST_LIST(/root/ansible-demo/ansible.cfg) = [u'/root/ansible-demo/hosts']
DEFAULT_LOG_PATH(/root/ansible-demo/ansible.cfg) = /var/log/ansible.log
# ansible demohost -om ping
demohost | SUCCESS => {"changed": false, "ping": "pong"}
# tail -n1 /var/log/ansible.log
2019-01-02 11:18:16,517 p=17106 u=root |  demohost | SUCCESS => {"changed": false, "ping": "pong"}
```

SE PUEDE HACER LA DEMO HASTA ESTE PUNTO PARA PROBAR EL FICHERO /var/log/ansible.log QUE REGISTRA TODAS LAS ACCIONES

Change the Default SSH Port and use SSH Compression

NO HACER ESTA PARTE DE LA DEMO. SE USA PARA CAMBIAR EL PUERTO SSH EN UN HOST COMO MECANISMO DE SEGURIDAD ADICIONAL

First we create a playbook that will help us modify the SSH port. For simplicity SELinux will be set to permissive (otherwise it would block the service from binding to the nonstandard port):

File: ssh-port.yaml

```
+ ---
+ - hosts: demohost
+   tasks:
+   - name: Set SSH port
+     lineinfile:
+       path: /etc/ssh/sshd_config
+       line: "Port {{ PORT }}"
+       regexp: '^Port '
+       insertafter: '^#Port '

+   - name: Switch SELinux state
+     selinux: state={{ STATE }} policy=targeted

+   - name: Restart sshd
+     service: name=sshd state=restarted
```

ESTE PLAYBOOK CAMBIA EL PUERTO SSH. PARA ELLO USA LAS VARIABLES PORT Y STATE AL INVOCAR EL PLAYBOOK

Do a trial run of the playbook passing the needed variables and verify the proposed change looks good, then make corrections or run for real:

```
# ansible-playbook -e PORT=2222 -e STATE=permissive --check --diff ssh-port.yaml
. . . snip . . .
TASK [Set SSH port] ********************************
--- before: /etc/ssh/sshd_config (content)
+++ after: /etc/ssh/sshd_config (content)
@@ -15,7 +15,7 @@
 # semanage port -a -t ssh_port_t -p tcp #PORTNUMBER
 #
 #Port 22
-Port 22
+Port 2222
 #AddressFamily any
 #ListenAddress 0.0.0.0
 #ListenAddress ::
. . . snip . . .
# ansible-playbook -e PORT=2222 -e STATE=permissive ssh-port.yaml
. . . snip . . .
demohost                   : ok=4    changed=3    unreachable=0    failed=0
```

NO EJECUTAR ESTOS COMANDOS DE PLAYBOOK

Try connecting to the host now that SSH is listening on the nonstandard port, then explore a variety of ways to fix the "problem":

```
# ansible demohost -om ping
```

```
demohost | UNREACHABLE!: Failed to connect to the host via ssh: ssh: connect to host station1.example.com port 22:
Connection refused # ansible --ssh-common-args "-o Port=2222" demohost -om ping
demohost | SUCCESS => {"changed": false, "ping": "pong"}
# echo "demo2host ansible_host=station1.example.com ansible_port=2222" >> hosts
# ansible demo2host -om ping
demo2host | SUCCESS => {"changed": false, "ping": "pong"}
# echo "remote_port=2222" >> ansible.cfg
# ansible demohost -om ping
demohost | SUCCESS => {"changed": false, "ping": "pong"}
```

Enable SSH Compression

File: ansible.cfg

```
  [defaults]
  log_path = /var/log/ansible.log
  inventory=hosts
  remote_port=2222
+ [ssh_connection]
+ ssh_args=-o ControlMaster=auto -o ControlPersist=60s -o Compression=yes
```

CUANDO SE CAMBIA EL PUERTO SSH ES NECESARIO CAMBIAR LA CONEXION EN EL FICHERO CONFIG. ESTE SERIA UN EJEMPLO

```
# ansible -vvv demohost -om ping
Using /root/ansible-demo/ansible.cfg as config file
META: ran handlers
Using module file /usr/lib/python2.7/site-packages/ansible/modules/system/ping.py
<station1.example.com> ESTABLISH SSH CONNECTION FOR USER: None
<station1.example.com> SSH: EXEC ssh -o ControlMaster=auto -o ControlPersist=60s -o Compression=yes -o Port=2222
```

Sneak Peek: Handlers, and Playbook Vars

The current playbook fails if vars are not passed, and restarts the SSH daemon regardless of if the config has changed. Here we embed default vars, and also make it idempotent (performs the restart only when needed) with a handler and notification:

File: ssh-port.yaml

```
   --
   - hosts: demohost
+    vars:
+      PORT: 22
+      STATE: enforcing
     tasks:
     - name: Set SSH port
   . . . snip . . .
         insertafter: '^#Port '
+    notify: Restart sshd
   . . . snip . . .

+  handlers:
     - name: Restart sshd
       service: name=sshd state=restarted
```

EJEMPLO DE PLAYBOOK CON EL USO DE VARIABLES Y HANDLERS (SE EXPLICA MÁS ADELANTE)

Change the port back to the standard one, and verify the playbook is idempotent by running it again and noting no changes:

```
# ansible-playbook ssh-port.yaml
demohost                   : ok=3    changed=3    unreachable=0    failed=0
# ansible-playbook ssh-port.yaml
demohost                   : ok=2    changed=0    unreachable=0    failed=0
```

Cleanup

```
# rm -f hosts ansible.cfg
```

Module Syntax Help

Ansible ships with over 1000 modules
Web docs include index:
http://docs.ansible.com/ansible/latest/modules_by_category.html
ansible-doc → detailed use info

- -l → list module name and description
- module_nameequal sign (=) indicates required argument
- -s → playbook snippets

Module Documentation

With over 1000 modules, Ansible has a huge number of systems that it can manage. You will naturally learn the names of the most commonly used modules and their arguments. For less commonly used modules, the documentation provides a quick reference. The web hosted docs are the most comprehensive and include an index, as well as links to further support.

From the CLI, Ansbile provides the **ansible-doc** command to provide help with module usage. To get a list of all modules along with a basic description, run **ansible-doc -l**. Because the list is so long, it is common to use **grep** to search the list for a key work or for a prefix associated with the system you wish to manage. For example:

```
$ ansible-doc -l | grep ^ec2_
ec2_ami       create or destroy an image in ec2
ec2_ami_copy  copies AMI between AWS regions, return new image id
ec2_ami_find  Searches for AMIs to obtain the AMI ID and other information
. . . snip . . .
```

For detailed help on a particular module, just pass the name of the module as the argument. Note that all required arguments will be indicated with an equal sign. The output will automatically be piped to a pager (less), and the end of the help includes examples of how the module is commonly called (in playbook syntax). For example:

```
$ ansible-doc file
> FILE    (/usr/lib/python2.7/site-packages/ansible/modules/files/file.py)

  Sets attributes of files, symlinks, and directories, or removes files/symlinks/directories. Many
  other modules support the same options as the `file' module - including [copy], [template], and
  [assemble].

Options (= is mandatory):

. . . snip . . .
- owner
```

```
        Name of the user that should own the file/directory, as would be fed to `chown'.  [Default: None]
= path
        path to the file being managed.  Aliases: `dest', `name' [Default: []]
```

To just get a list of the options with abbreviated descriptions use the -s (snippet) option. For example:

```
$ ansible-doc -s user
- name: Manage user accounts
  action: user
      append        # If `yes', will only add groups, not set them to just the list in `groups'.
      comment       # Optionally sets the description (aka `GECOS') of user account.
      createhome    # Unless set to `no', a home directory will be made for the user when the
                      account is created or if the home directory does not exist.
. . . snip . . .
```

Running Ad Hoc Commands

Execute one-off action against host(s)
ansible ***host_pattern [options]***
- -m *module_name* → default is command (can be changed in config)
- -a '*argument=value ...*' → check docs for list of required args

ansible-console
- interactive prompt for running commands on a host or group

Executing Ad-Hoc Commands

If an Ansible operation will be run many times, then it generally should be written as a playbook. However, on occasion, you may need to do something simple that you don't necessarily want to save for later. In these rare cases, using the **ansible** command to run an ad hoc command can be helpful. They can also be useful when you are first learning a particular module and want to quickly iterate until you have the options you need to include in your playbook.

The two most critical options are -m and -a which are used to specify the module to invoke and the arguments that will be passed to the module. If the module option is not specified, then the default module is used (command, or whatever is specified by the module_name config property). Consider the following examples:

Copy a config file to all hosts in the webhosts inventory group.

```
$ ansible webhosts -m copy -a "src=files/nginx.conf
      dest=/etc/nginx/
```

Remove the bryan user account (if it exists), from all hosts in the inventory (become root on remote host first):

```
$ ansible all -m user -a "name=bryan state=absent
      remove=yes" -b
```

Verify that Ansible can connect to the specified host as user guru and execute python with json lib. Output result on a single line:

```
$ ansible station1.example.com -m ping -o -u guru -k
SSH password: work Enter
station1.example.com | SUCCESS => {"changed": false,
      "ping": "pong"}
```

Poweroff all hosts in the hv group:

```
$ ansible hv -a 'poweroff' -b
```

Important Ad-Hoc Command Options

Running **ansible --help** will display the full list of options available. The following options are commonly used when running ad hoc commands:

-C or **--check** ⇒ don't make any changes; instead, try to predict some of the changes that may occur.
-D or **--diff** ⇒ when changing (small) files and templates, show the differences in those files.
-f or **--forks=** ⇒ specify number of parallel processes to use (default=5).
-i or **--inventory-file=** ⇒ specify inventory host path (default=/etc/ansible/hosts) or comma separated host list.
-t or **--tree=** ⇒ log output to this directory.
-u or **--user=** ⇒ connect as this user (default=current UID)
-k or **--ask-pass** ⇒ ask for connection password.
-b or **--become** ⇒ become another user (specify with -U) before executing commands (default=root).
-K or **--ask-become-pass** ⇒ ask for privilege escalation password.

ansible-console Command

If many ad hoc commands will be executed, the console allows easy invocation, and integrated documentation for the available modules. Invoke it passing any valid inventory pattern, and run commands against those hosts as needed. It includes an integrated command history that is retained even between sessions.

```
# ansible-console web
Welcome to the ansible console.
Type help or ? to list commands.

root@stations (6)[f:5]$ copy src=httpd.conf dest=/etc/httpd/conf/
. . . output omitted . . .
root@stations (6)[f:5]$ systemd name=httpd state=restarted
. . . snip . . .
```

DEMO: Running Ad Hoc Commands

Query systems to determine resource state
Install and configure software
Copy files to systems

Use console Ad Hoc Commands to List System Resources

```
[root@server1]# ansible-console stations
```

ANSIBLE CONSOLE PERMITE EJECUTAR COMANDOS AD-OC DE FORMA INTERACTIVA

```
[console]$ free -g
station4.example.com | SUCCESS | rc=0 >>
              total        used        free      shared  buff/cache   available
Mem:              3           0           3           0           0           3
Swap:             0           0           0
. . . snip . . .
[console]$ df -h /var
station2.example.com | SUCCESS | rc=0 >>
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/vg0-var  2.0G  220M  1.8G  11% /var
. . . snip . . .
[console]$ shell grep processor /proc/cpuinfo | wc -l
station2.example.com | SUCCESS | rc=0 | (stdout)
2
. . . snip . . .
[console]$ vgs
station2.example.com | SUCCESS | rc=0 >>
  VG  #PV #LV #SN Attr   VSize  VFree
  vg0   1   4   0 wz--n- 34.18g 22.68g
. . . snip . . .
[console]$ exit
```

 NO EJECUTAR ESTE BLOQUE DE COMANDOS EN LA SIGUIENTE PAGINA SE INDICA LO QUE SE DEBE EJECUTAR

Docker will need more space than the current volume. Grow the /var LV and filesystem and then verify:

```
# ansible stations -m lvol -a "vg=vg0 lv=var size=10G"
station2.example.com | SUCCESS => {
    "changed": true,
    "lv": "var",
    "size": 2,
    "vg": "vg0"
# ansible stations -m filesystem -a "dev=/dev/mapper/vg0-var fstype=xfs resizefs=yes"
station2.example.com | SUCCESS => {
    "changed": true,
    "msg": "meta-data=/dev/mapper/vg0-var. . . snip . . .
# ansible stations -a "df -h --output=size /var" -o
station1.example.com | SUCCESS | rc=0 | (stdout)  Size\n  10G
station2.example.com | SUCCESS | rc=0 | (stdout)  Size\n  10G
. . . output omitted . . .
```

Configure the needed repo and install the software on all hosts:

```
# ansible all -m yum_repository -a "name=Docker description='Docker repo' \
baseurl=http://server1/docker/ gpgcheck=no"
station4.example.com | SUCCESS => {
    "changed": true,
    "repo": "Docker",
    "state": "present"
}
# ansible all -m yum -a "name=docker-ce state=present"
. . . output omitted . . .
```

Create a simple config and copy it to the systems:

File: /root/ansible-demo/daemon.json

```
+ {
+ "insecure-registries": ["10.100.0.0/24"],
+ "registry-mirrors": ["http://server1:5001"],
+ "storage-driver": "devicemapper",
+ "storage-opts": ["dm.thinpooldev=/dev/mapper/vg0-dockerpool"]
+ }
```

```
# ansible stations -m file -a "name=/etc/docker state=directory"
# ansible stations -m copy -a "dest=/etc/docker/daemon.json src=daemon.json"
```

Create the referenced storage pool, add user to needed group, start service, and verify:

```
# ansible stations -m lvol -a 'vg=vg0 lv=dockerpool size=10G opts="-T"'
# ansible stations -m user -a "name=guru groups=docker state=present"
# ansible all -m systemd -a "name=docker enabled=yes state=started"
# ansible all -m file -a "path=/var/run/docker.sock mode=0666"
# ansible stations -a "docker version" -b --become-user guru
```

PARA INSTALAR DOCKER EN LA MAQUINA DE CONTROL Y LAS OTRAS SE PUEDE EJECUTAR EL SCRIPT install_docker.sh QUE SE IMPLEMENTA CON COMANDOS AD-OC O EL PLAYBOOK install_docker.yaml

./install_docker.sh

ó

ansible-playbook install_docker.sh

Dynamic Inventory

A program generates the inventory

- scripts already exist for many sources: Docker, AWS EC2, Openstack, etc.

inventory_script `--list` → **returns list of hosts and host groups as JSON hash**

inventory_script `--host` *hostname* → **returns JSON hash of host variables**

Dynamic Inventories

In cloud and virtualization environments, the number of hosts can change rapidly in response to orchestration events such as dynamic scaling in response to load. Even in largely static environments, the information about hosts may already be stored in some other inventory system and it may be desirable to produce an Ansible inventory on demand based on that information.

If the `ansible.cfg` inventory property, or the file specified by the `-i` CLI option is set executable, then Ansible will call the program expecting information in the form of a JSON hash to be returned. Dynamic inventory scripts must support being called with two options:

1. `--list` which must return the list of hosts and host groups.
2. `--host` *hostname* which must return a list of variables associated with the host.

Dynamic inventory scripts for most of the popular cloud and virtualization platforms already exist and can be found within the Github repo for Ansible. For example, the AWS EC2 inventory script can be downloaded from: https://raw.github.com/ansible/ansible/devel/contrib/inventory/ec2.py. Many of the inventory scripts will create host groups based on the meta-data shared by the hosts (think host variables). For example, in EC2, hosts can be assigned arbitrary tags, and all hosts with the same tag can be accessed through a group. For example:

```
$ ansible tag_webhosts -i inventory/ec2.py -m ping
```

Multiple Inventories

If the `ansible.cfg` lists the inventory parameter as a directory, then all executable files in the directory will be treated as dynamic inventory sources, and non-executable files will be used as normal static inventory sources.

ESTE BLOQUE DE DEMO CREA UNA SERIE DE IMAGENES EN LOCAL, LUEGO CREA UN REGISTRO EN EL NODO DE CONTROL Y LUEGO SUBE LAS IMAGENES AL REGISTRO NO EJECUTAR EL BLOQUE, HACER LO SIGUIENTE EN EL NODO DE CONTRO:

cd image_build
sudo ./docker-compose build
cd ..
sudo docker image ls
ansible-playbook registry_build.yaml
curl localhost:5000/v2/_catalog

DEMO: Dynamic Inventory

Create a Docker container infrastructure
Use a dynamic inventory script

Create a Docker Container Infrastructure

One of the many platforms that support dynamic Ansible inventory scripts is Docker. In order for the containers to be manageable by Ansible, they must have the basic Python components installed. First we build Docker images with Python included:

```
[root@server1 ansible-demo]# tar -xzf ~/GL380-course-files/demo-solutions/dynamic_inventory_demo.tar.gz
# systemctl enable --now docker
# cd image_build
# cat docker-compose.yaml # description of overall operations
. . . output omitted . . .
# cat centos7/Dockerfile  # build operations for this image
FROM centos:7
RUN  yum update -y && \
     yum install -y python && \
     yum clean all
# ./docker-compose build
. . . output omitted . . .
# docker image ls # view base and built images
POSITORY      TAG           IMAGE ID         CREATED            SIZE
u18-init      ansible       2d0ff78647f6     27 seconds ago     205MB
c7-init       ansible       69bb2ed55ca7     5 seconds ago      202MB
u18           ansible       3aacbd868713     11 seconds ago     121MB
c7            ansible       2e2182d405f1     7 minutes ago      204MB
ubuntu        18.04         1d9c17228a9e     5 days ago         86.7MB
centos        7             1e1148e4cc2c     4 weeks ago        202MB
```

Configure a Docker Registry to Host the Images

To make the newly built Docker images easily available to other hosts, we create a registry container and upload the images to it. The registry exposes a Docker compatible API for retrieving the images later. Instead of creating the playbook from scratch, we examine an existing one:

```
# cd ../container_build
# less registry.yaml
# ansible-playbook registry.yaml
. . . snip . . .
TASK [Create registry] *********************************************************************
fatal: [localhost]: FAILED! => {"changed": false, "failed": true, "msg": "Failed to import docker-py
      - No module named docker. Try `pip install docker-py`"}
        to retry, use: --limit @/root/ansible/docker/container_build/registry.retry

PLAY RECAP *********************************************************************************
localhost                  : ok=2    changed=0    unreachable=0    failed=1
```

The playbook caused an error because the Docker modules require additional Python modules to be installed on the Ansible control node. We fix the playbook by adding the tasks to install the missing component (Sneak peek - includes):

File: docker-py.yaml

```
+ ---
+ - name: Update Ansible Docker module
+   yum: name=python2-pip-8.1.2-5.el7.noarch state=present
+ - pip: name=docker-py
```

File: registry.yaml

```
  - hosts: localhost
    tasks:
+   - include: docker-py.yaml
    - name: Create registry
      docker_container:
```

```
# ansible-playbook registry.yaml
. . . output omitted . . .
# docker container ps
CONTAINER ID     IMAGE          ...  PORTS                                NAMES
b03acfdf96ac     registry:2.6   ...  5000/tcp, 0.0.0.0:5001->5001/tcp     registry-mirror
37569a490c9b     registry:2.6   ...  0.0.0.0:5000->5000/tcp               registry
# curl -I localhost:5000/v2/
HTTP/1.1 200 OK
Content-Length: 2
Content-Type: application/json; charset=utf-8
```

```
Docker-Distribution-Api-Version: registry/2.0
X-Content-Type-Options: nosniff
Date: Thu, 03 Jan 2019 23:03:09 GMT
```

The classroom server now has a registry and mirror running that can be used by the other Docker hosts in our lab. A final step is to load the images we need into that registry:

```
# ansible-playbook ~/GL380-course-files/demo-solutions/populate_reg.yaml
. . . snip . . .
TASK [Tag and push to registry] ************************************************************************
changed: [localhost] => (item={u'name': u'c7:ansible', u'repository': u'localhost:5000/centos:latest'})
changed: [localhost] => (item={u'name': u'u18:ansible', u'repository': u'localhost:5000/ubuntu:latest'})
changed: [localhost] => (item={u'name': u'c7-init:ansible', u'repository': u'localhost:5000/centos-init:latest'})
changed: [localhost] => (item={u'name': u'u18-init:ansible', u'repository': u'localhost:5000/ubuntu-init:latest'})

PLAY RECAP *********************************************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0
# curl localhost:5000/v2/_catalog
{"repositories":["centos","centos-init","ubuntu","ubuntu-init"]}
```

Start Containers on a Lab Host

Now we test the Docker infrastructure by having Ansible start containers on a managed system:

UNA VEZ CONFIGURADO EL REGISTRO LANZAR CONTENEDORES EN LOS NODOS
NOTA: ESTE PLAYBOOK INSTALA EL MÓDULO PYTHON PARA DOCKER A TRAVES DE PIP QUE ANTES DE LA V 2.6 DE PYTHON ERA docker-py Y DESPUES SE HA RENOMBRADO COMO docker

ansible-playbook container-cluster.yaml

NOTA: EN AWS ESTE PLAYBOOK FUNCIONA CORRECTAMENTE CON EL HOST localhost PERO FALLA CON LOS HOSTS DEL INVENTARIO (INCLUIDO EL LOCAL) INDICANDO QUE NO ENCUENTRA PIP. ES UN PROBLEMA CON QUE PYTHON3 NO TIENE UN MODULO ESPECIFICO PIP3 EN ANSIBLE A DIA DE HOY

https://github.com/ansible/ansible/issues/69028

File: /root/ansible-demo/container_cluster.yaml

```
+ ---
+ - hosts: station1.example.com
+   tasks:
+   - include: docker-py.yaml

+   - name: Start some Centos containers
+     docker_container:
+       name: "centos_{{ item }}"
+       image: server1:5000/centos
+       interactive: true
+       tty: true
+     with_sequence: count=3

+   - name: Start some Ubuntu containers
+     docker_container:
+       name: "ubuntu_{{ item }}"
+       image: server1:5000/ubuntu
+       interactive: true
+       tty: true
+     with_sequence: count=3
```

```
[server1]# cd ~/ansible-demo/
[server1]# cp ~/GL380-course-files/demo-solutions/docker-py.yaml .
[server1]# ansible-playbook container_cluster.yaml
. . . output omitted . . .
```

The playbook creates several containers on the demo station. Connect to the demo station and examine the results:

```
[station1]# docker container ps
CONTAINER ID      IMAGE                 COMMAND         CREATED             STATUS            PORTS    NAMES
aca60446d7e5      server1:5000/ubuntu   "/bin/bash"     27 seconds ago      Up 25 seconds              ubuntu_3
c45899dcdfd0      server1:5000/ubuntu   "/bin/bash"     28 seconds ago      Up 27 seconds              ubuntu_2
574a1aaa8710      server1:5000/ubuntu   "/bin/bash"     30 seconds ago      Up 28 seconds              ubuntu_1
0bfda71da72a      server1:5000/centos   "/bin/bash"     34 seconds ago      Up 33 seconds              centos_3
de5205088239      server1:5000/centos   "/bin/bash"     36 seconds ago      Up 35 seconds              centos_2
067c9c601023      server1:5000/centos   "/bin/bash"     38 seconds ago      Up 36 seconds              centos_1
```

Use an Ansible Dynamic Inventory Script

Now that the containers are running, we test the Docker dynamic inventory script:

```
[guru@station1]# mkdir ~/docker/; cd $_
[guru@station1]# wget https://raw.githubusercontent.com/ansible/ansible/devel/contrib/inventory/docker.py
[guru@station1]# chmod +x docker.py
[guru@station1]# ./docker.py --list
. . . output omitted . . .
[guru@station1]# ./docker.py --host centos_1
. . . output omitted . . .
```

docker.py FORMA PARTE DE ANSIBLE COMO INVENTARIO DINAMICO PERO NO FUNCIONA CON PYTHON3 QUE ES EL UNICO QUE FUNCIONA EN AWS

Finally, configure Docker to use the dynamic inventory and an appropriate transport:

File: ansible.cfg

```
+ [defaults]
+ inventory=docker.py
+ transport=docker
```

```
[guru@station1]# ansible all --list-hosts 2>/dev/null
  hosts (7):
    centos_2
    ubuntu3
    ubuntu1
    ubuntu2
    centos_1
    centos_3
    unix://var/run/docker.sock
[guru@station1]# ansible '~(centos|ubuntu)' -m ping
. . . output omitted . . .
```

Cleanup

```
[guru@station1]# docker container rm -f $(docker container ps -qa)
```

Lab 2

Estimated Time:
R7: 50 minutes

Task 1: Deploying Ansible [R7]

Page: 2-29 Time: 15 minutes

Requirements: 💻 (1 station)

Task 2: Ad Hoc Commands [R7]

Page: 2-37 Time: 20 minutes

Requirements: 💻 (1 station)

Task 3: Dynamic Inventories [R7]

Page: 2-43 Time: 15 minutes

Requirements: 💻 (1 station)

EL LABORATORIO 1 YA LO TENEMOS HECHO YA QUE HEMOS INSTALADO ANSIBLE

EL LABORATORIO 2 ES INTERESANTE PARA VER COMANDOS AD-OC

EL LABORATORIO 3 ES IDENTICO A LA ULTIMA DEMO Y EL INVENTARIO NO FUNCIONA CON PYTHON 3

Objectives

- Configure a system to be an Ansible control host (install Ansible)
- Understand and use Ansible host patterns
- Understand and edit the Ansible configuration file

Requirements

- (1 station)

Relevance

Lab 2
Task 1
Deploying Ansible [R7]

Estimated Time: 15 minutes

SI SE QUIERE INSTALAR ANSIBLE EN LAS OTRAS VM, HACERLO CON PYTHON3 PIP QUE LO INSTALA CON LAS DEPENDENCIAS DE PYTHON3

Installing Ansbile

1) The following actions require administrative privileges. Switch to a `root` login shell:

```
$ su -
Password: makeitso Enter
```

2) Configure the system to use the classroom server's repo where the Ansible packages are found:

```
# yum-config-manager --add-repo http://server1/ansible
Loaded plugins: langpacks
adding repo from: http://server1/ansible

[server1_ansible]
name=added from: http://server1/ansible
baseurl=http://server1/ansible
enabled=1
# rpm --import http://server1/ansible/RPM-GPG-KEY-EPEL-7
```

3) Install the core Ansible package and its dependencies:

```
# yum install -y ansible
. . . snip . . .
Installed:
  ansible.noarch 0:2.7.5-1.el7.ans

Dependency Installed:
  PyYAML.x86_64 0:3.10-11.el7   libyaml.x86_64 0:0.1.4-11.el7_0   python-babel.noarch 0:0.9.6-8.el7
  python-jinja2.noarch 0:2.7.2-2.el7   python-markupsafe.x86_64 0:0.11-10.el7   python-paramiko.noarch 0:2.1.1-0.9.el7
```

```
  sshpass.x86_64 0:1.06-1.el7

Complete!
```

4) Verify the version and try connecting to locahost (works without an inventory file or SSH since it uses local transport):

```
# ansible --version
ansible 2.7.5
  config file = /etc/ansible/ansible.cfg
  configured module search path = [u'/root/.ansible/plugins/modules', u'/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/lib/python2.7/site-packages/ansible
  executable location = /usr/bin/ansible
  python version = 2.7.5 (default, Aug  2 2016, 04:20:16) [GCC 4.8.5 20150623 (Red Hat 4.8.5-4)]
# ansible localhost -m ping
 [WARNING]: provided hosts list is empty, only localhost is available

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

Using Host Patterns

5) Configure Ansible to use the provided inventory file via environment variable, and test:

```
# export ANSIBLE_INVENTORY=/labfiles/inventory
# ansible all --list-host
 hosts (128):
    dc1-web01
    dc1-web02
    dc1-web03
    dc1-web04
. . . snip . . .
# ansible-inventory --graph
@all:
  |--@dbservers:
  |  |--dc1-db01
  |  |--dc1-db02
  |  |--dc1-db03
. . . snip . . .
```

ESTA PARTE ES INTERESANTE PROBARLA PARA DEMOSTRAR CÓMO SE GENERAN LAS LISTAS DE HOSTS DESDE EL INVENTARIO. EL FICHERO ESTA EN LA CARPETA DE LABORATORIOS ansible-labs/inventory

6) Use both the `ansible` and `ansible-inventory` commands with a host pattern to list all the systems in the first datacenter:

```
# ansible dc1 --list-hosts
  hosts (63):
    dc2-web01
    dc2-web02
    dc2-web03
    dc2-web04
. . . snip . . .
# ansible-inventory --graph dc1
@dc1:
  |--@dc1-rack01:
  |  |--dc1-db01
  |  |--dc1-db02
  |  |--dc1-web01
  |  |--dc1-web02
. . . snip . . .
```

7) Examine the provided inventory file to understand its organization, then create inventory patterns matching the described conditions. Test your patterns using either the `--list-hosts` option, or the `ansible-inventory` command to verify they work.

What pattern will match all webservers:

ansible webservers --list-hosts

Result: _____

What pattern will match all systems in rack 1 at datacenter 1:

Result: ansible dc1-rack01 --list-hosts

What pattern will match all systems in rack 1 at both datacenters:

Result: ansible dc1-rack01:dc2-rack01 --list-hosts

What pattern will match all systems at datacenter 1 that are not webservers:

Result: ansible 'dc1:!~web' --list-host

What pattern will match database servers at datacenter 1 in racks 1 and 3 (hint: do intersection last):

Result: ansible dc1-rack01:dc1-rack03 --list-hosts

What pattern will match dc1-web01 through dc1-web25, and dc2-web25 through dc2-web50 (hint: use slices):

Result: ansible 'webservers[0-24]:webservers[74-]' --list-hosts

LAS SLICES INDICAN EL ORDEN EN EL QUE SON LISTADOS LOS HOSTS EN CADA GRUPO CON ansible-inventory

What pattern will match both routers (hint: use REGEX):

Result: ansible '~router' --list-hosts

8) Clear the environment variable so that Ansible uses the default /etc/ansible/hosts inventory:

```
# unset ANSIBLE_INVENTORY
# ansible all --list-hosts
 [WARNING]: provided hosts list is empty, only localhost is available

 [WARNING]: No hosts matched, nothing to do

  hosts (0):
```

Edit Ansible Configuration

ESTA PARTE ES INTERESANTE PARA DEMOSTRAR CÓMO LOS FICHEROS DE CONFIGURACIÓN LOCALES TOMAN PRECEDENCIA

9) Examine the current Ansible configuration:

```
# ansible-config view
. . . output omitted . . .
# egrep -v '^(#|$)' /etc/ansible/ansible.cfg
[defaults]
[inventory]
[privilege_escalation]
[paramiko_connection]
[ssh_connection]
[persistent_connection]
[accelerate]
[selinux]
[colors]
[diff]
```

• Remove comments and blank lines to see real structure

10) Try running simple ad hoc commands without specifying the module, and note the results:

```
# cd
```

```
# ansible localhost -a "ls ~"
 [WARNING]: provided hosts list is empty, only localhost is available

localhost | SUCCESS | rc=0 >>
anaconda-ks.cfg
Desktop
original-ks.cfg
toppost.log

# ansible localhost -a "ls ~ | head -n1"
 [WARNING]: provided hosts list is empty, only localhost is available

localhost | FAILED | rc=2 >>
/root:
total 16
-rw-------. 1 0 0 2654 Jan 13  2017 anaconda-ks.cfg
drwxr-xr-x. 2 0 0   29 Jan 13  2017 Desktop
-rw-------. 1 0 0 1909 Jan 13  2017 original-ks.cfg
-rw-r--r--. 1 0 0 4763 Jan 13  2017 toppost.logls: cannot access |: No such file or directory
ls: cannot access head: No such file or directory
```

Second command fails because the default module is `command` which does not invoke the shell needed to process the pipe.

11) Create a config that modifies the default command to use the `shell` module instead, and test again:

File: ansible.cfg

```
+ [defaults]
+ module_name=shell
```

```
# ansible localhost -a "ls ~ | head -n1"
 [WARNING]: provided hosts list is empty, only localhost is available

localhost | SUCCESS | rc=0 >>
anaconda-ks.cfg
```

Now works because the new default invokes a shell which can process the pipe.

12) Create a stub playbook to test a few config options that only effect playbook runs (pay close attention to indentation and use spaces, not tabs):

Escriba el texto aquí

File: test.yaml

```
+ ---
+ - hosts: localhost
+   tasks:
+   - debug: msg="My first playbook"
```

13) Run the playbook to see the default behavior. Note that two tasks are executed (Gathering, and debug):

```
# ansible-playbook test.yaml
 [WARNING]: provided hosts list is empty, only localhost is available

PLAY [localhost] ***************************************************************************

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

TASK [debug] *******************************************************************************
ok: [localhost] => {
    "msg": "My first playbook"
}

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

14) Examine which config options pertain to the "cowsay" functionality, and their current settings. Then view the specific help for the 'COW_SELECTION' option needed to cycle through all stencils:

```
# ansible-config dump | grep COW
ANSIBLE_COW_PATH(default) = None
ANSIBLE_COW_SELECTION(default) = default
. . . snip . . .
# ansible-config list | grep -A8 COW_SELECTION:
ANSIBLE_COW_SELECTION:
  default: default
  description: This allows you to chose a specific cowsay stencil for the banners
    or use 'random' to cycle through them.
  env:
  - {name: ANSIBLE_COW_SELECTION}
  ini:
```

```
  - {key: cow_selection, section: defaults}
  name: Cowsay filter selection
```

15) Modify the config to use random cowsay templates, then install cowsay and test the playbook run again to see the results:

File: ansible.cfg

```
    [defaults]
    module_name=shell
+   nocows=0
+   cow_selection=random
```

```
# yum install -y cowsay
. . . output omitted . . .
# ansible-playbook test.yaml
< PLAY RECAP >
 ------------
  \
   \    (__)
        o o\
       ('') \---------
          \           \
           |           |\
           ||---(   )_|| *
           ||    UU  ||
           ==        ==

localhost                  : ok=2    changed=0    unreachable=0    failed=0
```

A random selection of animals now reports the name of each task and results.

16) Add another task to the playbook that is deliberately broken, run the playbook and note the behavior when it fails:

File: test.yaml

```
    tasks:
    - debug: msg="My first playbook"
+   - copy: src=foo dest=bar
```

```
# ansible-playbook test.yaml
```

```
. . . snip . . .
An exception occurred during task execution. To see the full traceback, use -vvv. The error was:
   If you are using a module and expect the file to exist on the remote, see the remote_src option
fatal: [localhost]: FAILED! => {"changed": false, "msg": "Could not find or access 'foo'\nSearched
   in:\n\t/root/files/foo\n\t/root/foo\n\t/root/files/foo\n\t/root/foo on the Ansible Controller.\n
   If you are using a module and expect the file to exist on the remote, see the remote_src option"}
        to retry, use: --limit @/root/test.retry
# ls *retry
test.retry
# ansible-config dump | grep -i ^retry
RETRY_FILES_ENABLED(default) = True
RETRY_FILES_SAVE_PATH(default) = None
```

When individual hosts fail a task, Ansible records which hosts failed allowing it to rerun with just the failed hosts. These retry files can get in the way, and config options exist to disable it entirely `retry_files_enabled`, or specify a different place to store them.

17) Configure Ansible to store retry files in `/tmp`, and test:

File: ansible.cfg

```
    [defaults]
    module_name=shell
    cow_selection=random
+   retry_files_save_path=/tmp
```

```
# rm -f *retry
# ansible-playbook test.yaml
. . . snip . . .
fatal: [localhost]: FAILED! => {"changed": false, "failed": true, "msg": "Unable to find 'foo' in expected paths."}
        to retry, use: --limit @/tmp/test.retry
```

Cleanup

18) Remove the config, test playbook, and cowsay package:

```
# rm -f ansible.cfg test.yaml
# yum remove -y cowsay
. . . output omitted . . .
```

Objectives

- Run ad-hoc commands with Ansible
- Explore CLI options to specify users, become users, and auth methods and passwords.

Requirements

- (1 station)

Relevance

Lab 2
Task 2
Ad Hoc Commands [R7]

Estimated Time: 20 minutes

Run Ad-Hoc Commands and Explore Auth Options

ESTOS COMANDOS AD-HOC SE HACEN EN STATION1 UNA VEZ INSTALADO ANSIBLE

1) Use an ad-hoc command to create a new user:

```
[root]# ansible localhost -m user -a "name=test shell=/bin/bash"
localhost | CHANGED => {
    "changed": true,
    "comment": "",
    "create_home": true,
    "group": 1002,
    "home": "/home/test",
    "name": "test",
    "shell": "/bin/bash",
    "state": "present",
    "system": false,
    "uid": 1002
}
```

CREA UN USUARIO TEST

2) Become the test user and try to install a package with an ad-hoc command:

```
[root]# su - test
[test]$ ansible localhost -m yum -a "name=cowsay state=present"
localhost | FAILED! => {
    "ansible_facts": {
        "pkg_mgr": "yum"
    },
    "changed": false,
    "msg": "You need to be root to perform this command.\n",
    "rc": 1,
    "results": [
        "Loaded plugins: langpacks\n"
```

CAMBIAR AL USUARIO TEST PARA PROBAR QUE NO PUEDE EJECUTAR COMANDOS SIN ELEVAR PRIVILEGIOS

TEST NO PUEDE INSTALAR UN PAQUETE DE SOFTWARE

```
    ]
}
```

Fails because yum must run as root for installing packages.

3) Try invoking something simple with the command module, and the become option, to see the "deeper" error:

PROBANDO CON BECOME TAMPOCO FUNCIONA PORQUE TEST NO ESTÁ APROBADO PARA ELEVAR PRIVILEGIOS EN /etc/sudoers

```
[test]$ ansible localhost -a "id" -b
localhost | FAILED! => {
    "changed": false,
    "failed": true,
    "module_stderr": "sudo: a password is required\n",
    "module_stdout": "",
    "msg": "MODULE FAILURE",
    "rc": 1
}
```

Fails this time because sudo requires a password.

4) Try again, this time using the option to prompt for become password (watch out, you NEED the upper case letter 'K', not lower case):

AUNQUE INTENTE HACER BECOME PREGUNTANDO LA PASSWORD TAMPOCO FUNCIONA PORQUE NO ESTÁ AUTORIZADO EN /etc/sudoers

```
[test]$ ansible localhost -a "id" -bK
SUDO password: work Enter
localhost | FAILED! => {
    "changed": false,
    "failed": true,
    "module_stderr": "Sorry, try again.\n[sudo via ansible, key=kjllenjrvgdpxtbgswsemlscyyplfykq] password: \n➥
   sudo: 1 incorrect password attempt\n",
    "module_stdout": "",
    "msg": "MODULE FAILURE",
    "rc": 1
}
```

Fails because even though the password has been passed, the test user is not allowed sudo access.

5) Try again, this time requesting use of su instead of sudo to become the root user:

```
[test]$ ansible localhost -m yum -a "name=cowsay state=present" -bK --become-method su
SU password: makeitso Enter
localhost | SUCCESS => {
    "changed": true,
    "msg": "",
    "rc": 0,
    "results": [
        "Loaded plugins: ...snip...Installed:\n  cowsay.noarch 0:3.04-4.el7    \n\nComplete!\n"
    ]
}
[test]$ cowsay "Ansible installed this program"
 ____________________________________
< Ansible installed this program >
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
[test]$ exit
```

CAMBIANDO A MÉTODO BECOME COMO su Y PREGUNTANDO LA PASSWORD
EL USUARIO TEST SÍ PUEDE INSTALAR EL SOFTWARE PORQUE CUALQUIER USUARIO PUEDE PASAR A su SIEMPRE Y CUANDO CONOZCA LA PASSWORD DE ROOT (CON SUDO QUE ES EL MÉTODO ESTANDAR NO NECESITA LA PASSWORD DE ROOT PERO DEBE SER AUTORIZADO EN /etc/sudoers

EL SOFTWARE coway SIMPLEMENTE MUESTRA EL MENSAJE ESCRITO CON UN ANIMAL

6) Try removing the package as the guru user both with and without the become option.

EL USUARIO GURU NO PUEDE QUITAR EL SOFTWARE SI NO ELEVA PRIVILEGIOS

```
[root]# su - guru
[guru]$ ansible localhost -m yum -a "state=absent name=cowsay"
 [WARNING]: provided hosts list is empty, only localhost is available

localhost | FAILED! => {
    "changed": false,
    "failed": true,
    "msg": "You need to be root to perform this command.\n",
    "rc": 1,
    "results": [
        "Loaded plugins: langpacks\n"
    ]
}
[guru]$ ansible localhost -m yum -a "state=absent name=cowsay" -b
 [WARNING]: provided hosts list is empty, only localhost is available

localhost | SUCCESS => {
```

COMO ESTÁ AUTORIZADO, CON LA OPCIÓN -b (BECOME) YA PUEDE QUITARLO

```
    "changed": true,
    "msg": "",
    "rc": 0,
    "results": [
        "Loaded plugins: ...snip...  cowsay.noarch 0:3.04-4.el7    \n\nComplete!\n"
    ]
}
[guru]$ exit
[root]# rm -f /etc/yum.repos.d/server1_ansible.repo
```

NO ES NECESARIO PORQUE LAS ESTACIONES NO USAN ESTE REPO

Works as guru with become option because guru is already permitted full sudo access.

7) Create another user account named test2 on the localhost using an Ansible ad hoc command. Use the documentation to review the module syntax as needed. The account should use bash as the shell, and have a home directory, but no password set.

CONSULTAR LA AYUDA CON ANSIBLE-DOC PARA HACER EL COMANDO AD-HOC PROPUESTO
ansible-doc -t module user

```
[root]# ansible-doc user
[root]# ansible Figure it out!
localhost | SUCCESS => {
. . . output omitted . . .
```

ansible localhost -m user -a "name=test2 shell=/bin/bash"

8) Verify that the account was created and can be used (via su):

```
[root]# su -c whoami test2
test2
```

9) Create a group named testgrp on the localhost using an Ansible ad hoc command.

```
[root]# ansible localhost -m group -a "Figure it out!"
```

ansible localhost -m group -a "name=testgrp"

10) Use an ad hoc command to modify the existing test2 account. It should add the user to the newly created testgrp (as a supplementary group), and also generate an SSH key that is encrypted with the passphrase secret.

```
[root]# ansible localhost -m user -a "Figure it out!"
localhost | SUCCESS => {
    "append": false,
```

ansible localhost -m user -a "name=test2 append=True groups="testgrp" generate_ssh_key=True"

```
"changed": true,
"comment": "",
"group": 1003,
"groups": "testgrp",
"home": "/home/test2",
"move_home": false,
"name": "test2",
"shell": "/bin/bash",
"ssh_fingerprint": "2048 d4:11:4a:bf:c6:41:4e:d0:9c:e1:08:35:28:ad:b7:76  ansible-generated on stationX.example.com (RSA)",
"ssh_key_file": "/home/test2/.ssh/id_rsa",
"ssh_public_key": "ssh-rsa AAAAB3Nza...snip...EhqyKjQx ansible-generated on stationX.example.com",
"state": "present",
"uid": 1003
```

11) Use an ad hoc command to attempt to set the password for the test2 user:

```
[root]# openssl passwd -1 -salt 123 secret
$1$123$S/xCJSfxKJIbRuK6dOfOi1
[root]# ansible localhost -m user -a "name=test2 password=$1$123$S/xCJSfxKJIbRuK6dOfOi1"
 [WARNING]: provided hosts list is empty, only localhost is available

localhost | SUCCESS => {
    "append": false,
    "changed": true,
    "comment": "",
    "group": 1003,
    "home": "/home/test2",
    "move_home": false,
    "name": "test2",
    "password": "NOT_LOGGING_PASSWORD",
    "shell": "/bin/bash",
    "state": "present",
    "uid": 1003
}
[root]# grep test2 /etc/shadow
test2:23/xCJSfxKJIbRuK6dOfOi1:17394:0:99999:7:::
```

LA CADENA ENCRIPTADORA DE LA PASSWORD SE GENERA CON EL COMANDO OPENSSL

PEGAR EN password= EL CONTENIDO DE LA CADENA ENCRIPTADA GENERADA CON OPENSSL

CON COMILLAS DOBLES LA CADENA ENCRIPTADORA NO SE INSERTA BIEN YA QUE LOS CARACTERES $ NO LOS INTERPRETA COMO LITERALES

Yikes, the password got mangled by the shell that thought the $1 was a variable and expanded it to nothing. The account is broken (effectively locked).

12) Try again, this time with the more protective single quotes to prevent attempts at variable expansion:

```
[root]# ansible localhost -m user -a 'name=test2 password=$1$123$S/xCJSfxKJIbRuK6dOfOi1'
. . . output omitted . . .
[root]# grep test2 /etc/shadow
test2:$1$123$S/xCJSfxKJIbRuK6dOfOi1:17394:0:99999:7:::
```

CON LOS ARGUMENTOS EN COMILLAS SIMPLES FUNCIONA CORRECTAMENTE YA QUE CONVIERTE TODO EN LITERAL

Now the password hash made it through correctly.

Cleanup

13) Remove the group and accounts using ad hoc commands executed via the Ansible console:

PARA ELIMINAR EL USUARIO Y EL GRUPO SE PUEDE PROBAR CON LA CONSOLA

```
[root]# ansible-console localhost
root@localhost (1)[f:5]$ user name=test state=absent
localhost | CHANGED => {
    "changed": true,
    "force": false,
    "name": "test",
    "remove": false,
    "state": "absent"
}
root@localhost (1)[f:5]$ user name=test2 state=absent
. . . output omitted . . .
root@localhost (1)[f:5]$ group name=testgrp state=absent
. . . output omitted . . .
root@localhost (1)[f:5]$ exit
```

Objectives

- Launch several Docker containers, and configure Ansible to manage them
- Use a dynamic inventory script instead of a static host inventory
- Run ad hoc commands with Ansible

Requirements

- (1 station)

Relevance

Lab 2
Task 3
Dynamic Inventories [R7]

Estimated Time: 15 minutes

ESTE LAB ES IDENTICO A LA DEMO DE INSTALAR DOCKER Y LANZAR CONTENEDORES PARA HACER EL INVENTARIO DINAMICO. EL INVENTARIO NO FUNCIONA CON PYTHON3

1) As the guru user, verify that Docker is installed, running, and configured with the correct insecure registry and mirror:

```
[guru]$ [ $(id -un) = "guru" ] || echo "STOP. Wrong user"
[guru]$ docker info | tail
. . . snip . . .
Insecure Registries:
 10.100.0.0/24
 127.0.0.0/8
Registry Mirrors:
 http://server1:5001/
Live Restore Enabled: false
```

NO ES NECESARIO VERIFICARLO. DOCKER YA ESTÁ INSTALADO

2) Use the provided playbook to start some containers based on the centos image, and verify:

```
$ ansible-playbook /labfiles/docker_run.yaml -b
. . . snip . . .
TASK [Start some "centos"containers]
changed: [localhost] => (item=1)
changed: [localhost] => (item=2)
changed: [localhost] => (item=3)
. . . snip . . .
$ docker container ps -q
cd867281b7c5
24d5b8582b16
45c90044e9a4
```

ESTE PLAYBOOKD CREA VARIOS CONTENEDORES DE DOCKER CON LOS QUE SE HACE LA PRUEBA DE INVENTARIO DINÁMICO

- These are randomly generated container IDs. We are just verifying there are three (indicating that three containers are running).

3) Download and test the Docker dynamic inventory script:

```
$ mkdir ~/docker/; cd ~/docker
$ cp /labfiles/docker.py .
$ chmod +x docker.py
$ ./docker.py --list
. . . output omitted . . .
$ ./docker.py --host centos_1
. . . output omitted . . .
```

docker.py ES UN CÓDIGO PYTHON QUE SE PUEDE BUSCAR EN GOOGLE Y GENERA UN INVENTARIO DINÁMICO CON LOS CONTENEDORES QUE ESTÁN PRESENTES EN UN HOST

- A full list of hosts/groups is returned along with all associated host variables as a JSON object.
- All host variables for specified host.

4) Configure Docker to use the dynamic inventory and an appropriate transport:

File: ansible.cfg

```
+ [defaults]
+ inventory=/home/guru/docker/docker.py
+ transport=docker
```

ESTE FICHERO DE CONFIGURACIÓN EN EL DIRECTORIO LOCAL TOMA PRECEDENCIA CON RESPECTO AL FICHERO DE CONFIGURACION GENÉRICO Y ESTABLECE QUE EL INVENTARIO SE HAGA CON docker.py HAY QUE CREARLO CON EL EDITOR nano O vi

5) Launch a few more containers, and then verify that Ansible can find them via the dynamic inventory and execute operations on them:

```
$ ansible-playbook -e image=ubuntu -e number=3 -b /labfiles/docker_run.yaml
$ ansible all --list-hosts 2>/dev/null
  hosts (7):
    centos_2
    ubuntu3
    ubuntu1
    ubuntu2
    centos_1
    centos_3
    unix://var/run/docker.sock
$ ansible '~(centos|ubuntu)' -m ping
. . . snip . . .
centos_1 | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
$ ansible '~(centos|ubuntu)_3' -a "cat /etc/os-release"
centos_3 | SUCCESS | rc=0 >>
NAME="CentOS Linux"
```

ESTE PLAYBOOK GENERA OTROS TRES CONTENEDORES BASADOS EN UBUNTU Y EL SEGUNDO COMANDO LISTA EL INVENTARIO USANDO EL INVENTARIO DINÁMICO DE docker.py

CASO DE USO DE DECLARACION EXTERNA DE VARIABLES

```
VERSION="7 (Core)"
ID="centos"
. . . snip . . .
ubuntu_3 | SUCCESS | rc=0 >>
NAME="Ubuntu"
VERSION="18.04.1 LTS (Bionic Beaver)"
ID=ubuntu
. . . snip . . .
```

6) View the groups automatically created by the dynamic inventory script:

```
$ ansible-inventory --graph
@all:
  |--@21dc339779002:
  |  |--centos_3
  |--@21dc3397790028ac19b194dc3e01c38cbaa40e4a3bad091b911a8f3f04dc50df:
  |  |--centos_3
. . . snip . . .
```

Cleanup

7) Remove the containers, but leave the config allowing Ansible to manage containers (used in later labs):

```
$ docker container rm -f $(docker container ps -qa)
175da9e2b73b
05bd9654a07b
76b669502587
cd867281b7c5
24d5b8582b16
45c90044e9a4
```

Content

Chapter 3

PLAYBOOKS BASICS

Writing YAML Files

Primary design goal: human readability
- structure primarily defined by indentation (spaces only NO tabs)

Strings
- generally not quoted, but can be (double or single quotes) for disambiguation

Lists
Hashes
Spanning lines
YAML Gotchas
- indentation / whitespace
- reserved characters: colon, curly brace, etc.

YAML (YAML Ain't Markup Language)

YAML was designed to be a human-friendly, cross language, Unicode based data serialization language with focus on the common native data types of agile programming languages. The following quote taken from the YAML specs provides a comparison to the JSON format and more insight into YAML design goals:

> "Both JSON and YAML aim to be human readable data interchange formats. However, JSON and YAML have different priorities. JSON's foremost design goal is simplicity and universality. Thus, JSON is trivial to generate and parse, at the cost of reduced human readability. It also uses a lowest common denominator information model, ensuring any JSON data can be easily processed by every modern programming environment.
>
> In contrast, YAML's foremost design goals are human readability and support for serializing arbitrary native data structures. Thus, YAML allows for extremely readable files, but is more complex to generate and parse. In addition, YAML ventures beyond the lowest common denominator data types, requiring more complex processing when crossing between different programming environments."
>
> — http://www.yaml.org/spec/1.2/spec.html

Data Types

Explicit data typing is seldom seen in the majority of YAML documents since parsers autodetect simple types. Depending on the implementation, YAML offers support for more complex "defined", or "user-defined" types, indicated with !!*defined_type_name value*, or !*user_type_name value* (See YAML specs for full details). Types can be declared to disambiguate if needed, for example:

```
a: 123          # an integer
b: "123"        # a string, disambiguated by quotes
c: 123.0        # a float
d: !!float 123 # also a float via explicit data type
e: !!str 123    # a string, disambiguated by explicit type
f: !!str True   # a string via explicit type
g: Yes          # a boolean True (yaml1.1), string "Yes" (yaml1.2)
```

YAML Lists

List constructs can be represented two ways in YAML. The first and most common is to specify the list name followed by a colon, and then indent each list item on the following lines starting the line with a dash. The amount that the list item lines are indented is not critical, but should be consistent and use spaces (not tabs or other characters). For example:

File: list_example.yaml

```
---
names:
  - Bryan
  - Sarah
  - Oscar
```

LA LISTA ES UNA ENUMERACION DE ELEMENTOS SIMPLES

The alternate, abbreviated format for lists is as follows:

File: list_example.yaml

```
---
names: ['Bryan', 'Sarah', 'Oscar']
...
```

YAML Hashes

Hashes, sometimes also called dictionaries, can also be represented two ways in YAML. Commonly seen is the longer format which uses key/value pairs indented to the same depth with a colon separating the key from value. For example:

File: hash_example.yaml

```
---
bryan:
  paternal haplogroup: R-L21
  maternal haplogroup: H1k
  neanderthal dna percent: 4
  neanderthal dna variants: 293
...
```

EL HASH O DICCIONARIO ES UNA ENUMERACIÓN DE ELEMENTOS CON UN VALOR

The alternate, abbreviated format for hashes is as follows:

File: hash_example.yaml

```
---
bryan: {paternal haplogroup: R-L21, maternal haplogroup:
       H1k, neanderthal dna percent: 4, neanderthal dna
      variants: 293}
...
```

Combination Data Structures

Most YAML files consist of combinations of these data structures. Variable data structures can get considerably more complicated than this at times. Consider the following example YAML which is a hash that also contains a list:

File: combo_example.yaml

```
---
bryan:
  age: 45
  species: homo sapien
  ancestry chromosome composition:
    - 23.9% French & German
    - 22.7% Scandinavian
    - 18.7% British & Irish
    - 11.4% Finish
    - 20.4% Broadly Northwestern European
    - 1.1% Broadly Southern European
    - 0.3% Native American
...
```

ES IMPORTANTE DISTINGUIR BIEN DONDE USAR UN HASH Y DONDE UNA LISTA EN LA SINTAXIS DE ANSIBLE

YAML Spanning Lines

Especially long string data can be made more readable by having it span multiple lines within the YAML file. There are nine different ways this can be done, each with their own use cases. Arguably the most common are literal scalars with clip (indicated by the pipe symbol |), and folded scalars with clip (indicated by the greater than symbol >). The following examples show these as they would appear in a task within an Ansible playbook:

File: literal_scalar_example.yaml

```
- name: Build the config
  copy:
    dest: /etc/the.config
    content: |
      This is the config data.
      It will contain multiple lines.
      It will end with a single new line.
      Even though there are several after this line.


- name: next task
```

LAS CADENAS MUY LARGAS SE PUEDEN INDICAR CON EL SIMBOLO | o >

File: folder_scalar_example.yaml

```
- name: Run complex command
  command: >
    command_name -opt 1
    -opt 2 -opt 3
    really_long_arg_name1
    arg2

- name: next task
```

LAS CADENAS MUY LARGAS SE PUEDEN INDICAR CON EL SIMBOLO | o >

LA DIFERENCIA ENTRE | Y > ES:
| INCLUYE LOS RETORNOS DE CARRO
> NO INCLUYE RETORNOS DE CARRO

YAML Gotchas

The first common error in YAML is incorrect indentation or indentation using tabs instead of spaces. Most modern editors have a way to make whitespace errors more visible either by making the characters visible, or via YAML specific syntax highlighting. Pick a capable editor like VIM and learn to use it properly.

The next most common error is using reserved structure characters without proper escaping. For example, the colon is used to separate key/value pairs in hashes, so it cannot be used unescaped at the end of a value:

File: bad_YAML.yaml

```
---
drive letter: c:
path: {{ base_name }}/dir1/file
```

File: good_YAML.yaml

```
---
drive letter: 'c:'
path: "{{ base_name }}/dir1/file"
```

Some uses of YAML such Ansible playbooks and docker-compose files include custom extensions which allow for use of variables within a YAML file. Be aware that the syntax for variables differs wildly across implementations.

Quotation marks are not strictly required for values, however, unquoted values will be interrupted by the YAML engine and exact rules vary by implementation. Further, the YAML values may also be interrupted by the tool running the YAML engine, such as Ruby.

The quoting rules are similar to those of a Linux shell. In general, special characters and newlines should all be in quotes. Much like shells in Linux, there is a difference between double-quotes and single-quotes. Double-quotes will interrupt some characters, such as, "\n" would be seen as a literal newline character. While a single-quoted '\n' would be seen as the string `\n`.

The words `Yes` or `No` should also be quoted, otherwise they will expand to Boolean true and false values. Whenever numbers are used in a value with characters such as a colon, minus sign, or plus sign quotes should also to prevent the YAML implementation from performing mathematical operations on the value.

Finally, when in doubt, process your YAML through a linter to see if it passes a structural test and perhaps to get it transformed into the canonical format: http://www.yamllint.com/.

Further resources for YAML syntax and usage can be found at: http://www.yaml.org/, http://docs.ansible.com/ansible/latest/YAMLSyntax.html, https://docs.docker.com/compose/compose-file/, https://en.wikipedia.org/wiki/YAML

REGLAS BÁSICAS MUY IMPORTANTES:
- NO USAR TABULADORES
- LA INDENTACIÓN ES IMPORTANTE
- LOS CARACTERES ESPECIALES QUE QUIERAN USARSE COMO LITERALES DEBEN ENTRECOMILLARSE

Playbook Structure

Playbook → list of plays
Play → hash of properties and task list
Task → module name
- hash of module args
- hash of properties

Playbook Structure

Playbooks consist of a list of plays. Plays typically begin with a `hosts:` hash, and then may have a list of variables and other control properties. The bulk of the play is typically the `tasks:` which are a list of modules to invoke along with their corresponding arguments. The following template playbook shows their structures:

File: playbook_template.yaml

```
---
# optional comments
- name: Demo playbook
  hosts: inventory_group
  property1: value
  property: value
  vars:
    var1: value
    var2: value
  tasks:
  - name: optional description of task
    module_name: module_arg1=value module_arg2=value

  - name: Task 2 description
    module_name:
      module_arg1: value
      module_arg2: value
...
```

Los names de los plays y las tasks aparecen durante la ejecución y sirven para ver en qué paso te encuentras

Las propertys indican una propiedad de un host (Sistema operativo, etc)

Las variables definidas aquí son valores por defecto pero se pueden cambiar al invocar el playbook

Las tareas tambien llevan nombre

Play Properties

Plays can include a hash of properties that effect operation. Roles, blocks, and tasks can also specify properties which supplement or override the play properties. A full list of supported properties is provided in the documentation: https://docs.ansible.com/ansible/latest/reference_appendices/playbooks_keywords.html Examples of commonly specified play properties include:

name ⇒ Identifier used for documentation
hosts ⇒ A list of groups, hosts or host pattern that translates into a list of hosts that are the play's target.
become ⇒ Boolean that controls if privilege escalation is used or not on Task execution. See also become_flags, become_method, become_user.
gather_facts ⇒ A boolean that controls if the play will automatically run the 'setup' task to gather facts for the hosts. See also gather_subset, and gather_timeout.
remote_user ⇒ User used to log into the target via the connection plugin.

Host and Task Execution Order

Inventory host pattern(s)
- default order is simply inventory order
- parallelism controlled by: `strategy`, `forks`, `serial`

Module execution returns data structure
- normally silently discarded
- can be displayed during execution(`ansible-playbook -v[v]...`)
- can be captured into variable (`register`)

Host and Task Execution Order

Each play includes inventory pattern(s) defined by the `hosts` property. The default strategy is "linear" and will cause each task in the task list to be executed, using an available thread (defined by `forks` (default 5). All hosts still in the `ansible_play_hosts` list must execute the task before any hosts are allowed to run the next task. Variances in host state will often cause task execution time to vary between hosts and thus the order of results to vary between play executions. If the strategy is switched to "free" then individual hosts can proceed to executing the next task as soon as they finish and a fork becomes available. This will allow an even greater variance in order.

The `serial` play property can batch this behavior to a subset of the hosts, which then run to completion of the play before the next batch starts. Starting with Ansible 2.2, serial can take lists and percentages in addition to simple integers. For details see: https://docs.ansible.com/ansible/latest/user_guide/playbooks_delegation.html#rolling-update-batch-size

The `order` play property effects the host processing order:

inventory ⇒ The default. The order is 'as provided' by the inventory
reverse_inventory ⇒ As the name implies, this reverses the order 'as provided' by the inventory
sorted ⇒ Hosts are alphabetically sorted by name
reverse_sorted ⇒ Hosts are sorted by name in reverse alphabetical order
shuffle ⇒ Hosts are randomly ordered each run

Normally if a host fails any task, it is removed from the `ansible_play_hosts` var and the remaining tasks in the play are not run for that host. If the playbook is idempotent and the changes "self contained", then simply fix the playbook and rerun. Blocks allow for more complex behavior on failure and are covered later. The following play properties can be used to further control if/when hosts are removed from the play list:

any_errors_fatal ⇒ Force any un-handled task errors on any host to propagate to all hosts and end the play.
ignore_errors ⇒ Boolean that allows you to ignore task failures and continue with play. It does not affect connection errors.
ignore_unreachable ⇒ Boolean that allows you to ignore unreachable hosts and continue with play. This does not affect other task errors (see ignore_errors) but is useful for groups of volatile/ephemeral hosts.

Although rare, sometimes it is necessary to run a task once only for a batch of hosts. The following task property can be used in those cases:

run_once: True ⇒ Boolean that will bypass the host loop, forcing the task to attempt to execute on the first host available and afterwards apply any results and facts to all active hosts in the same batch.

Note: `run_once` does not guarantee which host the task will run on. If that is important, then also add a `delegate_to:` *`inventory_host`*. Alternatively you can force a task to run only on specific hosts with a

when clause; for example: `when: inventory_hostname == rack1[0]` (execute only on the first host in the rack1 inventory group).

Beware: If `serial` is also set, then the `run_once` task is run for a host in each batch! Add the `when: inventory_hostname == ansible_play_hosts[0]` clause if it should run only on the first host in the first batch.

Return Values

Almost all Ansible modules return a data structure with results related to the module execution. Portions of this result will be displayed automatically when running ad-hoc commands, or when executing playbooks with the verbose options (`-v...`). Normally, this result is not retained any further, but if needed it can be kept by "registering" it into a variable. The result can then be referred to from other tasks within the play, or a portion sipmly printed to the screen using the `debug` module. The following documents the return values common to most modules:
http://docs.ansible.com/ansible/latest/common_return_values.html.

The following simple playbook shows printing return data to the screen. More complex use cases are found later in the course:

File: return_values_example.yaml

```
---
- hosts: localhost
  tasks:
    - name: Check health of GL site
      uri:
        url: http://www.gurulabs.com/_ping
        return_content: yes
      register: webpage
    - name: Display results
      debug:
        msg: "Code: {{ webpage.status }} Health: {{ webpage.content }}"
```

LA SALIDA DE UNA TASK SE PUEDE ALMACENAR EN UNA VARIABLE COMPLEJA TIPO HASH MEDIANTE register

```
# ansible-playbook /tmp/return_values_example.yaml
. . . snip . . .
TASK [Check health of GL site]
ok: [localhost]

TASK [Display results]
ok: [localhost] => {
    "msg": "Code: 200 Health: pong"
```

Command Modules

command
shell
script
expect
raw
idempotency
- simple: `creates|removes` parameters
- complex: `when:` clause based on previously registered result

Defining failure and overriding changed result
- `failed_when:`
- `changed_when:`

Command Modules

Most modules have a clearly defined set of tasks that they are targeted at. However, Ansible also ships with a number of general purpose modules that can run essentially arbitratry commands on the remote host. New users of Ansible often turn to these as an easy way to essentially take existing shell scripts or documented sequences of commands and very quickly create a playbook out of them. Over time, it is generally best to convert these type of playbooks to use the more targeted modules (primarily to ensure proper idempotence). If Ansible detects that command modules are invoking commands that likely overlap with functionality provided by other modules, it will issue a warning/recommendation when the playbook is executed.

There are 5 different command modules. The following describes their basic use, and shows sample tasks showcasing their use. Consult the online docs, or `ansible-doc` *`module_name`* for complete details:

command ⇒ executes the specified command directly (no pre-processing by shell). Default module for ad-hoc invocations.
shell ⇒ invokes a shell (default `/bin/sh`) and runs specified commands.
script ⇒ transfer the local script to the remote system and execute there. Python not required on the remote system.
expect ⇒ execute commands and respond to prompts (expect style).
raw ⇒ execute commands directly on the remote shell. Does not require Python on remote machine. Should generally be used only when device supports no other methods.

Note the use of a folded scalar to make the command arguments span mutiple lines for readability:

File: command_example.yaml

```
- command: >
    authconfig --update
    --enableldap --enableldapauth
    --ldapserver=server1.example.com
    --ldapbasedn='dc=example,dc=com'
    --enableldaptls
    --ldaploadcacert=http://server1.example.com/server.crt
```

COMMAND SOLO SOPORTA UN COMANDO SIMPLE DE UNA SOLA LINEA

Shell required because of the use of pipes and redirect. Note the use of the arg to set a working directory before execution:

File: shell_example.yaml

```
- shell: tune2fs -l /dev/VG1/data* | egrep '(Inode|Block
    |Reserved block) count' > fsreport
  args:
    chdir: /var/log/
```

SHELL PERMITE UNA LISTA DE COMANDOS O COMANDOS CONECTADOS CON PIPE (|)

Note that the script is a control node local file, and the working directory and environment on the remote side is determined by the Ansible connect user account properties:

SCRIPT PERMITE UN SCRIPT COMPLETO QUE SE TRANSFIERE DESDE LOCAL AL HOST GESTIONADO

File: script_example.yaml

```
- script: /playbooks/files/harden.sh --pci2.2 --auditing=max
```

Note the use of the modifer to make the prompt matching case insensitive:

File: expect_example.yaml

EXPECT SE USA CUANDO SE NECESITA UN COMANDO INTERACTIVO COMO UN LOGIN

```
- expect:
  command: "login {{ user }}"
  responses:
    (?i)password: "1nIt|@LpAs$"
```

Case sensitive password match

The following might be used to connect to a hypothetical network device supporting SSH (but not Python), and query it for temperature data:

File: raw_example.yaml

RAW es util para dispositivos con ssh pero sin soporte python (switches, routers, etc)

```
- raw: show env temp
```

Command Modules and Idempotency

Since the command modules can invoke any operation the host supports, detecting whether a command should be run is tricky. If the presence or absence of a file is a reliable way to determine if the command has run, then use of the `creates:` *`filename|fileglob`* or `removes:` parameters can easily be used to prevent the command from running on subsequent executions. For example:

File: idempotent_command_example.yaml

```
- command: make_config.sh
  creates: /etc/app/config
```

Command Modules and Error Handling

By default command modules will register an error if the return code from a command is not zero. The return code can be ignored entirely with the `ignore_errors: yes` parameter, or by directly controlling the return code; for example: `shell: example_commmand | true` but that should be avoided if at all possible. Assuming the returned data structure contains enough information to determine success or failure, better approach is to use the `failed_when` parameter and define the conditions; for example:

File: command_error_handling_example.yaml

```
- name: Examine stderr fail based on string match
  command: /usr/bin/the_command --parm
  register: comm_result
  failed_when: "'FAILED' in comm_result.stderr"

# diff exit status is 0 if inputs are the same,
# 1 if different, 2 if trouble.
- name: Fail task when both files are identical
  raw: diff file1 file2
  register: raw_result
  failed_when: diff_cmd.rc == 0 or diff_cmd.rc >= 2
```

The `changed_when` parameter can similarly override the "changed" result so it does not appear in the output report, or cause handlers to be notified.

COMO LOS MODULOS COMMAND, SHELL, ETC NO TIENEN IDEMPOTENCIA, failed_when O changed_when SE PUEDEN USAR COMO MECANISMO DE COMPROBACIÓN DE QUE HA TERMINADO CORRECTAMENTE O HA GENERADO EL CAMBIO ESPERADO

Significant Module Categories

Cloud Providers
Database
Monitoring
Network
Notification
Storage
Windows
General Sysadmin duties

- files
- packaging
- source control
- system

Cloud Providers

Modules that can interface with all of the most popular cloud providers, virtualization platforms, and container systems. Many of these are particularly powerful for provisioning and orchestartion tasks. For example in a disaster recovery situation, a provisioning playbook might connect to AWS and provision and entire virtual datacenter complete with VMs, storage, load-balancers, etc. Another playbook might be used to orchestrate the rolling update of a pool virtual web hosts within the Google compute cloud. Modules that connect to a variety of bare metal providers also exist, which might provision actual physical hardware on demand.

Database

With support for a large number of popular databases, this collection of modules can create/delete database, manage security with things like users and roles, and even control more complex operations such as replication.

Monitoring

These modules can connect to a large set of monitoring and logging services/devices. For example, a playbook might enable or disable alerts within Nagios for a set of systems mapped by dynamic Ansible inventory.

Network

Modules to interface with a huge number of network infrastructure devices. Playbooks might deploy new VLANs, create routes, enable DHCP on a LAN, update DNS records, configure network interface properties, create load-balancers, etc.

Notification

Modules to communicate with users and other systems. For example, after a provisioning playbook runs, it might send a collection of information about the deployment to users via email, rocketchat messages, irc, etc.

Storage

Modules to control a growing number of storage systems. Create storage volumes, along with their security rules and other properties. An exmple might be to connect to a Netapp and dynamically add LUNs as new VMs are provisioned.

Windows

Although Ansible is most popular within Linux based environments, it is rapdily growing into effectively managing heterogeneous environments that inclue Microsoft Windows systems. Many of the core systems administration modules once only available for Linux systems now have Windows equivalents.

File Manipulation

Copying files
- copy, fetch
- archive, unarchive
- synchronize

Editing files
- lineinfile, blockinfile
- replace
- patch

File properties
- stat
- file, acl, xattr

Copying Files

copia del controlador a los host gestionados

One common task within playbooks is the need to transfer files from one system to another. This is usually from the control server to the managed systems, but can also be in the other direction. Examples might include: copying a service config file to managed host, copying a key or certificate from the managed host back to the control node for use later in the playbook, or syncing up file share directories between several hosts.

Fetch / Copy

copia de los hosts gestionados al controlador

The following example playbook snippets show examples of using file copying modules. Here an web config is fetched from a remote managed host and then installed onto a group of hosts:

File: fetch_copy_example.yaml

```
- hosts: webmaster
  tasks:
  - name: grab good config
    fetch: dest=/tmp src=/etc/nginx/nginx.conf
```

Fetch trae un fichero de un servidor al sistema local

```
- hosts: webpool
  tasks:
  - name: install config on pool
    copy:
      src: /tmp/web1.example.com/etc/nginx/nginx.conf
      dest: /etc/nginx/
      mode: 644
```

Copy copia un fichero local a uno o varios servidores

Archive / Unarchive

The following example creates a compressed archive of some log directories on the managed host. Note that if compression is used like this, the compression program must exist on the target host.

File: archive_example.yaml

```
- name: Package logs
  archive:
    dest: /data/logs.tar.bz2
    format: bz2
    path:
    - /var/log/app*.log
    - /var/log/debug/
```

This playbook snippet would unpack an application's source onto the remote managed hosts. The module can also unpack an archive already on the remote host, or even download an archive from a URL and then unpack it to the remote host:

File: unarchive_example.yaml

```
- name: Deploy app from archive
  unarchive: src=/files/app1.tgz dest=/opt/app1
```

Synchronize (Rsync)

The synchronize module is a wrapper around the rsync command (which must be installed on both sides) that makes it easier to invoke rsync for common opertions. This modules is often used with the delegate_to operation to cause it to run on a specified remote host instead of the control host. The following simple example synchronizes a local copy of web content on the control host to the remote group of hosts:

File: synchronize_example.yaml

```
- synchronize:
    src: data/webcontent
    dest: /var/www/html/
```

Editing Files

Modifying the contents of existing files is a common systems administration need. The following examples demonstrate the use of core Ansible modules commoly used to edit files:

File: lineinfile_example.yaml

```
- name: add entry to hosts file
  lineinfile:
    path: /etc/hosts
    line: '192.168.2.42 foo.example.com'
    create: yes
```

File: blockinfile_example.yaml

```
- name: insert/update configuration and validate
  blockinfile:
    block: "{{ lookup('file', './local/ssh_config') }}"
    dest: "/etc/ssh/ssh_config"
    backup: yes
    validate: "/usr/sbin/sshd -T -f %s"
```

LOOKUP MIRA EL CONTENIDO DE UN FICHERO Y LO ASIGNA A UNA VARIABLE. EN ESTE CASO SE USA PARA INSERTAR EL CONTENIDO DEL FICHERO EN OTRO FICHERO

File: replace_example.yaml

```
- name: update DNS search path
  replace:
    dest: /etc/resolv.conf
    regexp:'^\s*search(.*)$'
    replace:'search\1 foo.com'
```

AÑADE LA RUTA FOO.COM A LA LINEA SEARCH (.*) CAPTURA LAS CADENAS EXISTENTES

Changing File Properties

Many modules that create, or edit the contents of files allow setting meta-data properties such as permission, ownership, and SELinux context as part of the operation. For changes effecting many files, or changing more obscure properties, or changing only meta-data (vs. also changing the data), specialized modules exist such as: file, acl, and xattr.

The stat module is somewhat unique in that it only retrieves and computes information related to the file and cannot modify the file in any way. Normally the data structure returned by this module is registered and later tasks are conditionally executed based on the data gathered.

Network Modules

nmcli
get_url
uri

Configuring Network Interfaces

There are many ways that network interfaces might be configured. For most Linux distributions, one option is to simply copy an appropriate config to the /etc/sysconfig/network-scripts/ (or equivalent) directory. This config can be constructed from a template on the control host. Another option is to use the command modules to invoke commands like **ip**, **route**, etc. directly (although this type of change is generally not persistent. The most common framework for managing networks in Linux today is the Network Manager system. The nmcli module is a wrapper around the **nmcli** command functions and can be used to manage network interface settings (including more advance configurations like interface teaming and bonding). The following example show configuring an interface with a static address:

File: nmcli_example.yaml

```
- nmcli:
    conn_name: dmz1
    ifname: eth1
    type: ethernet
    ip4: 192.0.2.100/24
    gw4: 192.0.2.1
    state: present
```

Download from URL

The get_url module allows downloading a file and then setting correct properties on it. This is commonly used anywhere you might have interactively used **wget** or **curl** to grab a file. The following example show downloading installation media, checking the hash to verify a good download, and then setting the needed file properties:

File: get_url_example.yaml

```
- name: Grab installation media
  get_url:
    url: http://mirror.example.com/isos/
      x86_64/CentOS-7-x86_64-Everything.iso
    dest: /export/isos
    checksum: sha256:b5bb9d8014a0f9b...812f4850b878ae4944c
    mode: 640
    setype: public_content_t
```

Interact with Web Services

A huge number of modern applications expose REST APIs. The Ansible uri module provides an easy way to interface with these services. It differs from the url module in that it can use any of the HTTP methods, can supply headers and data when posting, perform basic auth, follow redirects, and capture any returned header and data for further action. The following example, shows login into a web form, capturing a cookie that is returned, and using that to connect to a subsequent URL:

File: uri_example.yaml

```
- uri:
    url: https://example.com/login.php
    method: POST
    body: "name=bryan&password=secret&enter=Sign%20in"
    status_code: 302
    headers:
      Content-Type: "application/x-www-form-urlencoded"
  register: login

- uri:
    url: https://example.com/dashboard.php
    method: GET
    return_content: yes
    headers:
      Cookie: "{{login.set_cookie}}"
  register: dashresults
```

Packaging Modules

OSs

- package
- yum
- apt
- apk
- etc.

Langages

- pip
- npm
- etc.

Source control - git

OS Packaging Modules

One very common systems administration task is management of software. Initial provisioning of systems involves installing all the needed components, and sometimes ensuring certain packages are not installed. Then systems need ongoing patching as bugs and security vulnerabilities are discovered. Many different packaging systems exist within the Linux world each with different focus and capabilities. Ansible provides modules for the most common packaging systems like: Yum, APT, Zypper, APK, etc. It also has a generic module named package that provides an OS agnostic interface for the most common tasks. When creating more complicated playbooks in a heterogeneous environment, host facts can be used to detect the remote OS type and conditionally invoke OS specific package modules as needed.

Note that many package modules can be called with a list and will execute a single operation using the entire list. This will be covered later in the course. The following examples show common package management tasks:

File: package_example.yaml

```
- name: Install/update nginx
  package: name=nginx state=latest
```

File: yum_example.yaml

```
- name: Enable EPEL repo
  yum:
    name: https://dl.fedoraproject.org/pub/
      epel/epel-release-latest-7.noarch.rpm
    state: present

- name: Install Ansible
  yum: name=ansible state=present
```

Language Packaging Modules

Modern programing languages typically consist of a core set of functionality that can be extended by installing additional code modules/libraries/packages. Ansible provides modules to manage the language specific software from: Ruby, PHP, Perl, Python, Node, and others:

File: pip_example.yaml

```
- pip:
    requirements: /app1/requirements.txt
    virtualenv: /app1/venv
```

System Storage

parted
lvg, lvol
filesystem
mount

Managing Local Storage

Managing local storage is a common sysadmin duty that requires ongoing attention as application data sets expand or hosts are retasked. Ansible ships with many modules for direct storage management. The following example playbooks demonstrates creating a new partition, tagging it as a physical volume, extending an existing volume group, creating a new logical volume, and finally creating a filesystem:

File: partition_example.yaml

```
- name: Create new partition for LVM use
  parted:
    device: /dev/sdb
    number: 2
    flags: [ lvm ]
    state: present
    part_start: 10GiB
    part_end: 500Gib
```

File: storage_example.yaml

```
- name: Create PV
  shell: pvs | grep -q /dev/sdb2 || pvcreate /dev/sdb2

- name: Create or resize VG
  lvg: vg=vg0 state=present pvs=/dev/sda2,/dev/sdb2

- name: Create app_data LV
  lvol: vg=vg0 lv=app_data size=100G

- name: Create app_data filesystem
  filesystem: dev=/dev/mapper/vg0-app_data fstype=xfs
```

Once filesystems have been added to the system, they still need to be mounted on bootup. The following playbook could be used to inject the needed line into the /etc/fstab file and then test it by mounting the filesystem:

File: mount_example.yaml

```
- name: Mount filesystem
  mount:
    path: /data1
    fstype: xfs
    src: /dev/mapper/vg0-demo
    state: mounted
```

Account Management

user
group
authorized_keys

Account Management

User and group system accounts can be easily managed by the core user and group modules. The user module is especially powerful and can additionally create an SSH key for the account, and update the password. The following example shows creating groups and accounts, and then installing an SSH key as trusted:

File: accounts_example.yaml

```
---
- hosts: localhost
  vars:
    teamleads: ['red', 'blue', 'green']
  tasks:
  - name: Create groups
    group: "name={{ item }}"
    with_items: "{{ teamleads }}"

  - name: Add team lead accounts
    user:
      name: "{{ item }}lead"
      shell: /bin/bash
      generate_ssh_key: yes
      group: users
      groups: "{{ item }}, wheel"
    with_items: "{{ teamleads }}"
```

```
# id redlead
uid=1002(redlead) gid=100(users) groups=100(users)
      ,10(wheel),1002(red)
# id bluelead
uid=1003(bluelead) gid=100(users) groups=100(users)
      ,10(wheel),1003(blue)
```

File: ssh_authorized_key_example.yaml

```
  - name: Set authorized key
    authorized_key:
      user: "{{ item }}lead
      state: present
      key: "{{ lookup('file', '/files/id_rsa.pub') }}"
    with_items: "{{ teamleads }}"
```

Security

selinux*
iptables
firewalld
ufw

Security

One major reason people adopt configuration management tools like Ansible is to ensure that systems are compliant with security policies. Once a policy has been described in the form of a playbook, it is easy to audit systems and bring them into compliance if/when they drift from the required configuration state. Security is a very broad topic and nearly all Ansible modules can play a part. In terms of major security frameworks that protect the system overall, SELinux and Netfilter are arguably the most significant for most Linux systems. SELinux is complex and correspondingly has multiple Ansible modules supporting it: seboolean, sefcontext, selinux, selinux_permissive, and seport. A discussion of how to use SELinux is beyond the scope of this course.

Netfilter (the kernel level packet filtering for Linux), currently has several different user-space systems for managing the in kernel packet manipulation rules. Ansible has modules corresponding to the older style iptables framework, and the newer ufw and firewalld dameons.

The following example playbook shows configuring a service to listen on a non-standard port and then adding the necessary SELinux and Netfilter rules to allow the service to operate on that new port:

File:

```
- name: Set SSH port
   lineinfile:
     path: /etc/ssh/sshd_config
     line: "Port {{ PORT }}"
     regexp: '^Port '
     insertafter: '^#Port '
   notify: Restart sshd

- seport:
   ports: 2002
   proto: tcp
   setype: ssh_port_t
   state: present

- name: Firewall allows alternate SSH inbound
  firewalld:
     zone: dmz
     port: 2002/tcp
     permenant: true
     state: enabled
```

The following example uses the iptables module to modify the INPUT chain policy and then add rules to accept traffic from a list of hosts defined in the jumphosts variable:

File: basic_firewall_example.yaml

```
- name: Reject traffic by default
  iptables:
    chain: INPUT
    policy: DROP

- name: Accept traffic from all jumphosts
  iptables:
    chain: INPUT
    source: "{{ item }}"
    jump: ACCEPT
  with_items: "{{ jumphosts }}"
```

Services

service, runit, svc, systemd

- enable/disable
- start/stop/restart/reload

sysctl

- what kernel features are enables
- how they are tuned

System Modules

A booted system generally has many services running. Some of these services are kernel threads, and others are running in user-space. The kernel exposes writable data structures in via the procfs virtual filesystem that can be used to enable, disable, or tune the operation of many internal kernel processes. The Ansible sysctl module provides the interface to set these properties and maipulate the files they are normally stored in. Most user-space processes are started and managed by isome other service. Historically, this was the init binary, but other more sophisticated systems exist like daemontools, runit, and the modern Linux systemd. Ansible has modules for all of these major service management systems and also offers the generic service module.

Note that one reason to use the more specific module (for example systemd vs just service) is if you need access to the more complete, systemd specifc, data structure returned by the module (for example details about the services CGroup property limits).

The following examples show setting some kernel tuning options and manipulating services:

File: kernel_tune_example.yaml

```
- sysctl:
    name: net.ipv4.ip_forward
    value: 1
    sysctl_set: yes
```

File: service_restart_example.yaml

```
- name: Stop webserver
  systemd: state=stopped name=httpd

- name: Restart db
  systemd: state=restarted name=mariadb

- name: Start webserver
  systemd: state=started name=httpd
```

DEMO: Playbooks

Provision additional storage

Provision Additional Storage

Ansible has many modules related to provisioning storage. The following demo show what it is like to attach new storage to a system. First let's query the disk via an ad hoc command to see what partitions exist on it and what the returned data-structure looks like:

```
[guru@station1]$ mkdir ~/playbooks
[station1]$ ansible localhost  -m parted↵
      -a "state=info device=/dev/sda unit=s" -b
localhost | SUCCESS => {
    "changed": false,
    "disk": {
        "dev": "/dev/sda",
        "logical_block": 512,
        "model": "ATA QEMU HARDDISK",
        "physical_block": 512,
        "size": 104857600.0,
        "table": "msdos",
        "unit": "s"
    },
    "failed": false,
    "partitions": [
        {
            "begin": 2048.0,
            "end": 1026047.0,
            "flags": [
                "boot"
            ],
            "fstype": "xfs",
            "name": "",
            "num": 1,
            "size": 1024000.0,
            "unit": "s"
        },
        {
            "begin": 1026048.0,
            "end": 72706047.0,
            "flags": [
                "lvm"
            ],
            "fstype": "",
            "name": "",
            "num": 2,
            "size": 71680000.0,
            "unit": "s"
        }
    ],
    "script": "unit 's' print" }
```

Now, using the information about where the existing partitions start and end, we can create a playbook that creates a new third partition. For new partition start sector, use the current last partition end sector plus 1. For new partition end sector, use new start + 521000:

```
[station1]$ echo $((72706048 + 512000))
73218048
```

estas demos están en solutions/chap3 se pueden probar en la vm de control o en otras vm

File: ~/playbooks/create_partition.yaml

```
+ ---
+ - hosts: localhost
+   tasks:
+   #warning: assumes a specific disk structure!
+   - name: Create third partition
+     parted:
+       device: /dev/sda
+       number: 3
+       flags: [ lvm ]
+       state: present
+       part_start: 72706048s
+       part_end: 73218048s
```

We run the playbook first in check mode and with verbose options to see what it is about to do:

```
[station1]$ ansible-playbook -C ~/playbooks/create_
      partition.yaml -b -vv
. . . snip . . .
changed: [localhost] => {..."script": "unit KiB mkpart
      primary 72706048s 73218048s unit KiB set 3 lvm on"}
```

Assuming the test run looks good, we run it for real:

```
[station1]$ ansible-playbook ~/playbooks/create_
      partition.yaml -b -vv
. . . output omitted . . .
```

To get the storage into use, we can create a new VG, LV, and filesystem, and then mount it:

File: ~/playbooks/create_VG_LV_FS.yaml

```
+ ---
+ - hosts: localhost
+   tasks:
+   # warning, assumes specific disk arrangment
+   - name: Create VG
+     lvg: vg=vg1 state=present pvs=/dev/sda3
+
+   - name: Create demo LV
+     lvol: vg=vg1 lv=demo size=100%FREE
+
+   - name: Create demo filesystem
+     filesystem: dev=/dev/mapper/vg1-demo fstype=xfs
+
+   - name: Mount filesystem
+     mount: path=/demo fstype=xfs
         src=/dev/mapper/vg1-demo state=mounted
```

```
[station1]$ ansible-playbook ~/playbooks/create_partition.yaml -b
. . . output omitted . . .
```

Sneak Peek - Conditionals

Right now the playbook will fail if it is run and the VG has no free space to use. Ideally we want to be able to run the playbook at any time and know the outcome will be the state described. In this case, we need the create LV task to be skipped if the VG has no space. This can be accomplished with a `when:` clause:

File: ~/playbooks/create_VG_LV_FS.yaml

```
    - name: Create demo LV
      lvol: vg=vg1 lv=demo size=100%FREE
+     when:
+       - ansible_lvm.vgs.vg1.free_g != "0"
```

Lab 3

Estimated Time:
R7: 100 minutes

Task 1: Playbook Basics [R7]

Page: 3-24 Time: 20 minutes

Requirements: 💻 (1 station)

Task 2: Playbooks: Command Modules [R7]

Page: 3-31 Time: 20 minutes

Requirements: 💻 (1 station)

Task 3: Playbooks: Common Modules [R7]

Page: 3-39 Time: 60 minutes

Requirements: 💻 (1 station)

Objectives

- Writing and debugging YAML
- Using common Ansible Modules
- Debugging playbooks

Requirements

- (1 station)

Relevance

Lab 3
Task 1
Playbook Basics [R7]

Estimated Time: 20 minutes

LA TASK UNO ES PARA REVISAR LA SINTAXIS DE PLAYBOOKS
SI DA TIEMPO HACERLA, EN CASO CONTRARIO PASAR A LA TASK2

1) As the guru user, examine and the try running the first of the provided sample working playbook:

```
[guru]$ cd ~
$ cat /labfiles/working1.yaml
---
- hosts: localhost
  tasks:
  - name: Simple greeting
    debug: msg="Hello from the playbook!"
$ ansible-playbook /labfiles/working1.yaml
. . . snip . . .
TASK [Simple greeting]
ok: [localhost] => {
    "msg": "Hello from the playbook!"
}
. . . snip . . .
```

Note how the module args are listed on the same line using the equal sign.

2) Examine and run the second working playbook:

```
$ cat /labfiles/working2.yaml
---
- hosts: localhost
  tasks:
  - name: Simple greeting
    debug:
      msg: "Hello from the playbook!"
$ ansible-playbook working2.yaml
. . . snip . . .
```

```
TASK [Simple greeting]
ok: [localhost] => {
    "msg": "Hello from the playbook!"
}
. . . snip . . .
```

Note how the module args are listed on separate indented lines as a YAML hash using the colon.

3) Try running the broken1 playbook that mixes the two formats seen above:

```
$ cp /labfiles/broken* ~
$ ansible-playbook broken1.yaml
ERROR! Syntax Error while loading YAML.

The error appears to have been in '/root/ansible/labfiles/broken1.yaml': line 6, column 14, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

    copy: dest=/tmp/broken1
      content: "Will this work?"
             ^ here
```

4) Choose either of the two workable syntaxes and edit the `broken1.yaml` playbook fixing it. Then test:

```
$ editor broken1.yaml
$ ansible-playbook broken1.yaml
. . . snip . . .
TASK [copy config]
changed: [localhost]

PLAY RECAP
localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

• Use your editor of choice...

PARA PODER EDITAR LOS FICHEROS HAY QUE COPIARLOS DEL DIRECTORIO /labfiles AL DIRECTORIO DEL USUARIO. SE PUEDE USAR NANO O VI COMO EDITOR

5) Examine and run third, more complicated, working playbook:

```
$ cat /labfiles/working3.yaml
. . . output omitted . . .
$ ansible-playbook /labfiles/working3.yaml
. . . snip . . .
```

```
localhost                  : ok=7    changed=4    unreachable=0    failed=0
$ ls -l /tmp/working3
-rw-r-----. 1 guru guru 65 Aug 21 15:15 /tmp/working3
$ cat /tmp/working3
This is not really several lines of text once it hits the file.
```

Spend a moment really looking at the syntax for this playbook. This will help you spot the errors that are introduced next.

6) Look at the broken playbook and see if any errors stand out to you:

```
$ cat broken3.yaml
. . . output omitted . . .
```

7) Try running a broken version of the previous playbook to discover the first error produced:

```
$ ansible-playbook broken3.yaml
ERROR! Syntax Error while loading YAML.

The error appears to have been in '/labfiles/broken3.yaml': line 13, column 5, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

    This message is longer
    it spans a few lines
    ^ here
```

Lines within a folded scalar must be indented further than the element that proceeds them.

8) Edit the file and indent the three indicated lines correctly:

File: broken3.yaml

```
    copy:
      dest: /tmp/greeting
      content: |
→       This message is longer
→       it spans a few lines
→       and uses fancier yaml
```

9) Try running again to discover the next error the parser encounters:

```
$ ansible-playbook broken3.yaml
. . . snip . . .
The offending line appears to be:

  - name: Wait a few seconds
    pause: prompt=hang on: seconds=3
                         ^ here

exception type: <class 'yaml.scanner.ScannerError'>
exception: mapping values are not allowed here
  in "<unicode string> line 17, column 26:
        pause: prompt=hang on: seconds=3
```

YAML meta characters can't be used as literal characters unless they are escaped properly (usually within quotes).

10) Place the string within quotes to fix the error:

File: broken3.yaml

```
   - name: Wait a few seconds
→     pause: prompt="hang on:" seconds=3
```

11) Try running again to discover the next error the parser encounters:

```
$ ansible-playbook broken3.yaml
. . . snip . . .
The offending line appears to be:

        This is not really
        several lines of text
^ here
There appears to be a tab character at the start of the line.
. . . snip . . .
```

Here the error message is pretty clear. No tabs for indentation in YAML! Remember that the `cat -T` ***filename*** command can be used to make tab visible as ^I to make it easier to find them. Many editors have a similar function. For example in Vim use: `:set list`

12) Edit the line, removing the tab character and indenting it to the proper depth using spaces instead:

File: broken3.yaml

```
    line: >
        This is not really
→       several lines of text
        once it hits the file.
      path: /tmp/working3
```

13) Try running again to discover the next error the parser encounters:

```
$ ansible-playbook broken3.yaml
. . . snip . . .
ERROR! this task 'debug' has extra params, which is only allowed in the following modules: command,
   win_command, shell, win_shell, script, include, include_vars, include_tasks, include_role, import_tasks,
   import_role, add_host, group_by, set_fact, raw, meta

The error appears to have been in '/labfiles/broken3.yaml': line 5, column 5, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

  tasks:
  - name: Simple greeting
    ^ here
```

The parser thinks that the words "from", "the", and "playbook" are parameters to the debug module instead of a string containing the value of the msg parameter.

14) Fix the issue by adding quotes so the value is properly read as a string:

File: broken3.yaml

```
  tasks:
  - name: Simple greeting
→   debug: msg="Hello from the playbook!"
```

15) Try running again to discover the next error the parser encounters:

```
$ ansible-playbook broken3.yaml
. . . snip . . .
ERROR! no action detected in task. This often indicates a misspelled module name, or incorrect module path.

The error appears to have been in '/labfiles/broken3.yaml': line 26, column 5, but may
be elsewhere in the file depending on the exact syntax problem.

The offending line appears to be:

  - name: Add greeting to file
    ^ here
```

The parser is looking for the name of an Ansible module for this task and doesn't recognize "linesinfile".

16) Edit the file and fix the module name (be sure to maintain correct indentation):

File: broken3.yaml

```
  - name: Add greeting to file
→ ~~  linesinfiles~~lineinfile:
      line: >
```

17) Try running again to discover the next error the parser encounters:

```
$ ansible-playbook broken3.yaml
. . . snip . . .
TASK [Create empty file]
fatal: [localhost]: FAILED! => {"changed": false, "failed": true, "msg": "Unsupported parameters for (file)➥
   module: perms Supported parameters include: attributes,backup,content,delimiter,diff_peek,directory_mode➥
   ,follow,force,group,mode,original_basename,owner,path,recurse,regexp,remote_src,selevel,serole,setype,➥
   seuser,src,state,unsafe_writes,validate"}
[WARNING]: Could not create retry file '/labfiles/broken3.retry'.         [Errno 13] Permission denied: u'➥
   /labfiles/broken3.retry'
```

The file now passes basic YAML parsing, and also high level module description parsing. This error is not encountered until it goes to run this specific task and finds a parameter it does not recognize for this module. If you encounter errors

like this, you would probably consult the output of `ansible-doc` ***`module_name`*** to find the correct parameter name.

18) Edit the file and fix the parameter name:

File: broken3.yaml

```
    - name: Create empty file
→     file: path=/tmp/working3 state=touch ~~perms~~mode=640
```

19) Run the playbook one final (hopefully) time. This time it should complete without errors:

```
$ ansible-playbook broken3.yaml
. . . output omitted . . .
```

Objectives

- Use the script module
- Use the command and shell modules
- Use the raw module

Requirements

- (1 station)

Relevance

Lab 3
Task 2
Playbooks: Command Modules [R7]

Estimated Time: 20 minutes

ESTE LAB ES INTERESANTE PARA VER LA DIFERENCIA ENTRE EJECUTAR UN SCRIPT BASH Y EJECUTAR LO MISMO EN UN PLAYBOOK

Script Module

HACERLO CON EL USUARIO GURU PARA COMPROBAR QUE NO PUEDE EJECUTAR UN SCRIPT SIN ELEVAR PERMISOS

1) As the guru user, create a directory to work in and examine the script that will be run:

```
[guru]$ mkdir ~/playbooks
[guru]$ cd ~/playbooks
[guru]$ cat /labfiles/mkwebdir.sh
. . . output omitted . . .
```

EL SCRIPT CREA UN USUARIO Y UN DIRECTORIO CON UNA PAGINA WEB BÁSICA

Look through the script logic and make sure you understand what it will do when called with a username as a positional argument.

2) Create a simple playbook that uses the script module to execute this on your local system:

File: webdir_script.yaml /labfiles/solutions/chap3/webdir_script.yaml

```
+ ---
+ - hosts: localhost
+   tasks:
+   - name: Create user web dir
+     script: /labfiles/mkwebdir.sh visitor
```

3) Try running the playbook (as the guru user):

```
[guru]$ ansible-playbook webdir_script.yaml
. . . snip . . .
TASK [Create user web dir]
fatal: [localhost]: FAILED! => {"changed": true, "failed": true, "msg": "non-zero return code", "rc": 1, "stderr":
```

```
"mkdir: cannot create directory '/home/visitor/public_html': Permission
denied\n/home/guru/.ansible/tmp/ansible-tmp-1503425491.12-31315115973340/mkwebdir.sh: line 9:
/home/visitor/public_html/index.html: Permission denied\nchown: cannot access '/home/visitor/public_html':
Permission denied\nchmod: changing permissions of '/home/visitor': Operation not permitted\n", "stdout": "",
"stdout_lines": []} to retry, use: --limit @/home/guru/playbooks/webdir_script.retry
```

Guru does not have permission to enter the visitor user's home directory, so script execution fails.

4) Execute again, this time using the become option:

```
[guru]$ ansible-playbook webdir_script.yaml -b
. . . snip . . .
localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

Works fine due to becoming root first.

5) Execute again and note that the "changed" column still reports a change:

```
[guru]$ ansible-playbook webdir_script.yaml -b
. . . snip . . .
localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

This is not good because nothing really changed. The problem is that Ansible doesn't understand that the script has already run and that the task can be skipped. The proper fix is to define a check so that the task only executes when it needs to.

6) Modify the playbook adding an arg so the task will be skipped if a file it creates already exists, then test again to verify no change is reported:

File: webdir_script.yaml — editarlo con nano para añadir la línea

```
    - name: Create user web dir
      script: /labfiles/mkwebdir.sh visitor
+     args:
+       creates: /home/visitor/public_html/index.html
```

```
[guru]$ ansible-playbook webdir_script.yaml -b
```

CREATES SIRVE PARA IMPLEMENTAR LA IDEMPOTENCIA EN UN MODULO SCRIPT YA QUE COMPRUEBA QUE YA EXISTE LO QUE TIENE QUE HACER.

```
. . . snip . . .
localhost                  : ok=1    changed=0    unreachable=0    failed=0
```

One weakness of using the script is that parts of it could still fail without Ansible detecting that. As long as the final command returns zero as its exit code, and the file is created, Ansible will think that everything is fine. Splitting the tasks down into individual commands can make Ansible more aware of each step and able to monitor it.

Command and Shell Modules

7) Create a playbook that accomplishes the same thing as the last one, but uses the command and shell modules as appropriate to execute the operations:

File: /home/guru/playbooks/webdir_commands.yaml

```
+ ---
+ - hosts: localhost
+   tasks:
+   - name: Find homedir
+     shell: "getent passwd visitor | cut -d: -f6"
+     register: HOMEDIR

+   - name: Create webdir
+     command: "mkdir -p {{ HOMEDIR.stdout }}/public_html"

+   - name: Create webpage
+     shell: >
+       echo "<html><h1>Hello World</h1></html>"
+       > "{{ HOMEDIR.stdout }}/public_html/index.html"

+   - name: Set ownership
+     command: "chown -R visitor: {{ HOMEDIR.stdout }}/public_html/index.html"

+   - name: Set permissions
+     command: "chmod 755 {{ HOMEDIR.stdout }}"
```

/labfiles/solutions/chap3/webdir_commands.yaml.v1

8) Back out the changes from running the script previously and then test the new playbook:

```
[guru]$ sudo rm -rf /home/visitor/public_html/
```

```
[guru]$ sudo chmod 700 /home/visitor/
[guru]$ ansible-playbook webdir_commands.yaml -b
. . . snip . . .
localhost                  : ok=6    changed=5    unreachable=0    failed=0
```

Note that the design of the current playbook will cause many tasks to execute when they need not, and report changes when none actually occur. This is bad, and a hint that we may not be taking the right approach. Arguably the best fix in this case is to refactor using Ansible modules other than command and shell for most of the work. An alternative is to provide checks and conditions to help Ansible correctly execute tasks only when needed and report finding correctly.

Bonus: (if time permits)

9) Expand the playbook to properly execute tasks only when needed and report changes correctly:

File: webdir_commands.yaml

```
   ---
   - hosts: localhost
     tasks:
     - name: Find homedir
       shell: "getent passwd visitor | cut -d: -f6"
       register: HOMEDIR
+      changed_when: false

     - name: Create webdir
       command: "mkdir -p {{ HOMEDIR.stdout }}/public_html"
+      args:
+        creates: "{{ HOMEDIR.stdout }}/public_html"

     - name: Create webpage
       shell: >
         echo "<html><h1>Hello World</h1></html>"
         > "{{ HOMEDIR.stdout }}/public_html/index.html"
+      args:
+        creates: "{{ HOMEDIR.stdout }}/public_html/index.html"

+    - name: Check ownership
+      command: "stat -c %U {{ HOMEDIR.stdout }}/public_html/index.html"
+      register: OWNER
+      changed_when: false

     - name: Set ownership
       shell: "chown -R visitor: {{ HOMEDIR.stdout }}/public_html/index.html"
+      when: OWNER.stdout != "visitor"

+    - name: Check permission
+      command: "stat -c %a {{ HOMEDIR.stdout }}"
+      register: PERM
+      changed_when: false

     - name: Set permissions
       command: "chmod 755 {{ HOMEDIR.stdout }}"
+      when: PERM.stdout != "755"
```

/labfiles/solutions/chap3/webdir_commands.yaml.v2

changed_when: false nunca reporta que ha cambiado algo aunque lo haya hecho

when indica cuándo debe realizarse la task, si no se cumple la condición no se ejecuta

10) Try running the playbook several times and note that it now skips tasks when appropriate and reports stats correctly:

```
[guru]$ ansible-playbook webdir_commands.yaml -b
. . . snip . . .
TASK [Check ownership]
ok: [localhost]

TASK [Set ownership]
skipping: [localhost]

TASK [Check permission]
ok: [localhost]

TASK [Set permissions]
skipping: [localhost]

PLAY RECAP
localhost                  : ok=6    changed=0    unreachable=0    failed=0
```

Raw Module Execution

Trabaja con un dispositivo raw (el contenedor alpine no tiene de serie python)
ES OPCIONAL HACERLO SEGUN EL TIEMPO QUE SE TENGA

11) Switch to the ~/docker directory created in the earlier lab and start a simple container:

```
[guru]$ cd ~/docker
[guru]$ docker container run -dti --name raw_test alpine
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
88286f41530e: Pull complete
Digest: sha256:1072e499f3f655a032e88542330cf75b02e7bdf673278f701d7ba61629ee3ebe
Status: Downloaded newer image for alpine:latest
3a7d58419c8c9442da4e1ce848c5b2641e667444448ea171a1cb722e7581b21b
```

12) Try using the ping module to verify Ansible can connect to the container and run operations:

```
[guru]$ ansible -m ping raw_test
. . . snip . . .
raw_test | FAILED! => {
    "changed": false,
    "module_stderr": "/bin/sh: /usr/bin/python: not found\n",
```

```
    "module_stdout": "",
    "msg": "MODULE FAILURE\nSee stdout/stderr for the exact error",
    "rc": 127
}
```

None of the normal Ansible modules will work because the host does not have the prerequisite Python installed.

13) Try executing a command on the host using the raw module:

```
[guru]$ ansible -m raw -a "ip addr li eth0" raw_test
. . . snip . . .
raw_test | SUCCESS | rc=0 >>
49: eth0@if50: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
```

Works because this does not require Python on the managed host.

14) Create a simple playbook that can be used to bootstrap Ansible managing a host by installing Python using the raw module:

File: bootstrap.yaml

```
+ ---
+ - hosts: ~raw
+   gather_facts: false
+   tasks:
+   - name: Get Python installed
+     raw: "apk update; apk add python"
+   - name: Test connection
+     ping:
```

/labfiles/solutions/chap3/bootstrap.yam
Está modificado añadiento apk upgrade por que si no no funciona

Note that this particular code would only work for systems that use the apk package system.

15) Execute the playbook and verify that Ansible can now execute modules on the host:

```
[guru]$ ansible-playbook bootstrap.yaml --limit raw_test
. . . snip . . .
PLAY [all]

TASK [Get Python installed]
changed: [raw_test]

TASK [Test connection]
ok: [raw_test]

PLAY RECAP
raw_test                   : ok=2    changed=1    unreachable=0    failed=0
```

--limit altera el target del playbook para ejecutarlo solo en los hosts indicados

Cleanup

16) Remove the container:

```
[guru]$ docker container rm -f raw_test
raw_test
```

Objectives

- Modify file properties
- Copy and archive files
- Edit files
- Administer accounts
- Install software, and manage services
- Manage firewall rules

Requirements

- (1 station)

Relevance

Lab 3
Task 3
Playbooks: Common Modules [R7]

Estimated Time: 60 minutes

Speed Tip (optional)

1) Create a small script (new playbook) that can provide the boilerplate when making new playbooks:

```
[guru]$ mkdir ~/bin
$ touch ~/bin/np; chmod 755 ~/bin/np
```

File: /home/guru/bin/np

```
+ #!/bin/sh
+ cat << END > $1.yaml
+ ---
+ - hosts: localhost
+   tasks:
+   - name:

+ ...
+ END
+ vim $1.yaml
```

Es un generador de plantillas para crear scripts con la infraestructura y sintaxis mínima. No es necesario hacerlo. En cualquier caso está en /labfiles/solutions/chap3

Create new playbooks by running: `np playbook_name`.

Note: feel free to replace the final line with an invocation of whatever your preferred editor is.

File Properties

Los playbooks propuestos en el resto del lab son para pensarlos. Las soluciones están en /labfiles/solutions/chap3
Pensar primero lo que se haría y luego ver el código y testearlo. para tenerlos todos operativos hacerlo siguiente:

```
cd /home/guru/docker
cp /labfiles/solutions/chap3* .
```

2) Create a playbook named `properties.yaml` that uses the "file", and "acl" modules to meet the following conditions:

- Creates the `/tmp/file1` file with
- permissions of `u=rw,g=r`
- others must have no permissions to the file
- owned by `guru`
- group of `users`
- an ACL that grants the `visitor` user read access

- Creates a `/tmp/file2` sym-link that points to `/tmp/file1`

/labfiles/solutions/chap3/properties.yaml

o=

Hint: read ansible-doc page for file module: state=file and state=touch

3) Execute your playbook and then grade it by running the provided playbook:

```
$ ansible-playbook properties.yaml -b
. . . output omitted . . .
$ ansible-playbook /labfiles/properties_grade.yaml
. . . snip . . .
TASK [debug]
ok: [localhost] => {
    "msg": "Nice job, you pass."
}
PLAY RECAP
localhost                  : ok=8    changed=0    unreachable=0    failed=0
```

If the grading does not indicate you pass and instead fails on an assertion, then recheck your playbook, fix, and repeat grading.

Copy and Archive Files

4) Create two containers needed for the playbooks in the next exercise:

```
[guru]$ cd ~/docker
$ ansible-playbook -e number=2 /labfiles/docker_run.yaml
. . . snip . . .
TASK [Start some centos containers]
changed: [localhost] => (item=1)
changed: [localhost] => (item=2)
. . . snip . . .
```

el playbook docker_run.yaml crea el numero de contenedores que se le indiquen por defecto usa la imagen centos

5) Create a stub playbook that has two plays. One that creates a directory on the local system to store the files, and the other that targets the centos hosts:

File: ~/docker/fetch_copy1.yaml

está ubicado en /labfiles/solutions/chap3 y contiene todo el código incluido la propuesta del punto 6

```
+ ---
+ - hosts: localhost
+   gather_facts: false
+   tasks:
+   - name: create empty directory to hold hosts files
+     file: path=/tmp/hosts state={{ item }}
+     loop: ["absent", "directory"]

+ - hosts: centos_1,centos_2
+   tasks:
```

gather_facts: false inhibe la ejecución implicita del módulo "setup" que al principio de cada playbook extrae los facts de cada host. Esto se hace para acelerar la ejecución cuando estos facts no son necesarios

Ejemplo de loop sobre una lista de elementos, que será comentado en un capítulo posterior

6) Extend the second play of the stub playbook so that it uses the "fetch" and "copy" modules to meet the following conditions:

- The /etc/hosts file from each centos host is copied to the local /tmp/hosts directory.
- The files in /tmp/hosts are named based on the Ansible inventory hostname of the managed hosts; for example /tmp/hosts/centos_1. (Hint: see examples from ansible-doc fetch)
- The entire local /tmp/hosts directory is copied back to all centos host's /tmp directory.
- The /tmp/hosts/README file exists on all centos hosts and contains "hosts files from all centos containers" (line must also end with a newline).

7) Execute your playbook and then grade it by running the provided commands and verifying the output matches your systems output:

```
$ ansible-playbook ~/docker/fetch_copy1.yaml
. . . output omitted . . .
$ docker container exec centos_1 ls /tmp/hosts
README
centos_1
centos_2
$ docker container exec centos_1 cat /tmp/hosts/README
hosts files from all centos containers
```

docker container exec ejecuta un comando dentro del container

```
$ docker container exec centos_1 grep centos /tmp/hosts/*
/tmp/hosts/centos_1:172.17.0.2  centos_1
/tmp/hosts/centos_2:172.17.0.3  centos_2
```

8) Create a new playbook named ~/docker/archive1.yaml that uses the "archive" module to meet the following conditions: /labfiles/solutions/chap3/archive.yaml

- Contains a single play that runs on both centos hosts.
- Creates an archive named /tmp/backup.tgz on each centos hosts that contains /etc/hostname and /etc/os-release.
- Archive is a gzip compressed tar.

9) Execute your playbook and then grade it by running the provided commands and verifying the output matches your systems output:

```
$ ansible-playbook ~/docker/archive1.yaml
$ docker container exec centos_1 tar tzvf /tmp/backup.tgz
-rw-r--r-- root/root         9 2017-08-23 09:12 hostname
-rw-r--r-- root/root       393 2016-11-29 11:12 os-release
$ docker container exec centos_2 tar tzvf /tmp/backup.tgz
-rw-r--r-- root/root         9 2017-08-23 09:12 hostname
-rw-r--r-- root/root       393 2016-11-29 11:12 os-release
```

Edit Files

10) Create a new playbook named ~/docker/edit1.yaml that uses the "lineinfile", and "replace" modules to meet the following conditions: /labfiles/solutions/chap3/edit1.yaml

- The /etc/passwd file has: "guru:x:1000:100::/tmp:/bin/bash" as the final line.
- Switch /sbin/nologin to /bin/false for each instance within /etc/passwd

11) Execute your playbook and then grade it by running the provided commands and verifying the output matches your systems output:

```
$ ansible-playbook ~/docker/edit1.yaml
$ docker container exec centos_1 cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
```

```
adm:x:3:4:adm:/var/adm:/bin/false
lp:x:4:7:lp:/var/spool/lpd:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/bin/false
operator:x:11:0:operator:/root:/bin/false
games:x:12:100:games:/usr/games:/bin/false
ftp:x:14:50:FTP User:/var/ftp:/bin/false
nobody:x:99:99:Nobody:/:/bin/false
systemd-bus-proxy:x:999:998:systemd Bus Proxy:/:/bin/false
systemd-network:x:192:192:systemd Network Management:/:/bin/false
dbus:x:81:81:System message bus:/:/bin/false
guru:x:1000:100::/tmp:/bin/bash
```

Administer accounts

12) Create a new playbook named `~/docker/account1.yaml` that uses the "user", and "group" modules to meet the following conditions:

/labfiles/solutions/chap3/account1.yaml

- The `dialout` group is removed.
- The `admin` group is created with gid of 13.
- The `demo` account is created, with uid of 2000, a home directory of `/home/demo`, and shell of `/bin/sh`.
- The `demo` account primary group is `admin`, and is also a member of the `wheel`.

13) Execute your playbook and then grade it by running the provided commands and verifying the output matches your systems output:

está mal el playbook es account1.yaml

```
$ ansible-playbook ~/docker/edit1.yaml
$ docker container exec centos_1 tail -n1 /etc/passwd
demo:x:2000:13::/home/demo:/bin/sh
$ docker container exec centos_1 egrep '(wheel|admin)' /etc/group
wheel:x:10:demo
admin:x:13:
```

Install Software and Manage Services

14) Remove the previous containers, and then use the provided script to start a container named web_1:

```
$ docker container rm -f centos_{1,2}
$ sh /labfiles/docker_centos_init.sh web_1
```

este script genera un container con systemd presente que permite instalar servicios.

Containers normally only run one process, and not an init system. This image is specially configured with systemd present, and the container is running with options necessary for the operation of systemd within a container. A full discussion of these complexities is beyond the scope of this course, thus the script to launch the container for us.

15) Copy the playbook from `/labfiles` and use it as a starting point for your playbook.

/labfiles/solutions/chap3/software.yaml

```
$ cp /labfiles/software1.yaml ~/docker/
```

Edit the playbook adding tasks that use the "yum", "yum_repository", and "systemd" modules to meet the following conditions:

- Configures a new repo with baseurl: "http://10.100.0.254/centos7", and gpgkey: "http://10.100.0.254/centos7/RPM-GPG-KEY-CentOS-7"
- Installs the `iproute`, and `httpd` packages.
- Ensures `httpd` it is configured to start on boot and also currently started.
- Verifies a connection to the new web server returns HTTP status 200. (already DONE by provided stub)

16) Execute your playbook and then grade it by confirming the provided "Verify connections" output reports ok, and "Connection results" are seen:

```
$ ansible-playbook ~/docker/software1.yaml
. . . snip . . .
TASK [Verify connections]
ok: [web_1 -> localhost]

TASK [Connection results]
ok: [web_1] => {
    "msg": "\"URL: http://172.18.0.2/\n Returned: <html>Welcome to container: c2601f35b8b8</html>\"\n"

}
PLAY RECAP
web_1                      : ok=7    changed=0    unreachable=0    failed=0
```

El playbook está completo con la propuesta y la solución

Es interesante ver en el playbook la opción delegate que ejcuta una tarea en un host distinto de la lista

17) Use the existing software playbook to configure a second web container:

```
$ sh /labfiles/docker_centos_init.sh web_2
$ ansible-playbook ~/docker/software1.yaml
. . . snip . . .
PLAY RECAP
web_1                      : ok=7    changed=0    unreachable=0    failed=0
web_2                      : ok=7    changed=4    unreachable=0    failed=0
```

Manage Firewall Rules

18) Create and run the following playbook that modifies the Netfilter input chain policy to drop packets by default:

File: firewall.yaml

```
---
- hosts: ~web
  gather_facts: No
  tasks:
  - name: Drop by default
    iptables:
      chain: INPUT
      policy: DROP
```

/labfiles/solutions/chap3/firewall.yaml

```
$ ansible-playbook firewall.yaml
. . . output omitted . . .
```

19) Add additional tasks to the playbook that uses the same module to meet the following conditions:

- Connections to TCP/80 are accepted.
- ALL ICMP is accepted.

20) Run the playbook and verify it works using the following commands:

```
$ ansible-playbook firewall.yaml
. . . output omitted . . .
$ docker exec web_1 iptables -L INPUT
Chain INPUT (policy DROP)
```

```
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             anywhere             tcp dpt:http
ACCEPT     icmp --  anywhere             anywhere
```

Cleanup

21) Remove the web containers:

```
$ docker container rm -f web_{1,2}
web_1
web_2
```

Content

Chapter 4

VARIABLES AND INCLUSIONS

Variables

Hold data that can then be referenced in a play

- can be defined in many places
- scope can be: global, play, task, or host

Referenced via Jinja2 syntax

Complex order of precedence

- generally does "the right thing"
- pass via command line for ultimate win

Variables

Variables in Ansible store values that can then be used within plays. They can be defined in several ways, and have different scope depending on where they are defined. Variables allow playbooks to be more adaptable as their values can easily be changed to match the current environment.

Variable names always begin with a letter (upper or lower case), and then have some number of letters, numbers, or underscores. They cannot contain other special characters (like [.*?$":;] etc.).

No matter how the variable is defined, it is accessed the same way; using Jinja2 templating syntax. For example: "The variable my_var is equal to {{ my_var }}".

Dictionary Variables

Dictionary Variables can be set inside a playbook using YAML syntax:

File: example.yaml

```
vars:
  field1: one
  field2: two
```

These are accessed with either bracket notation: {{ vars['field1'] }}

Or dot notation: {{ vars.field1 }}

Fun Fact: On the python layer, the steps are slightly different between the two notation formats (dot vs bracket):

For foo.bar:

1. Check for an attribute called 'bar' on 'foo': getattr(foo, 'bar')
2. If no attribute, check from an item 'bar' in 'foo': foo.__getitem__('bar')
3. If no item, return an undefined object

For foo['bar']:

1. Check for an item 'bar' in 'foo': foo.__getitem__('bar')
2. If no item, check for attribute called 'bar' on 'foo': getattr(foo, 'bar')
3. If no attribute, return an undefined object

Variable Precedence

There may be situations where variables are defined in multiple ways. In these cases, it is helpful to know which order the variables will be processed. The order of precedence (Ansible version 2.7.5) is:

- command line values (eg "-u user")
- 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)
- include_vars
- set_facts / registered vars
- role (and include_role) params
- include params
- extra vars (always win precedence)

To ensure your variable is always given precedence, you can pass it on the command line as a parameter:

```
$ ansible-playbook test.yaml --extra-vars⏎
      "version=1.23.45 some_other_variable=something"
```

Variables - Playbooks

vars: **per task or play**
vars_files: **per play**
include_vars: **task**

Defining Variables - Playbooks

Inside your YAML playbook, you can assign variables that directly relate to a particular play. These will be scoped to that play and not accessible from other plays. For example:

File: variable.yaml

```
---
- hosts: localhost
  vars:
   my_var: "Hello"
  tasks:
  - debug: msg="my_var is {{ my_var }}"

- hosts: localhost
  tasks:
  - debug: msg="my_var is {{ my_var }}"
```

```
$ ansible-playbook /tmp/variable.yaml
. . . snip . . .
TASK [debug]
ok: [localhost] => {
    "msg": "my_var is Hello"
}

PLAY [localhost]

TASK [debug]
fatal: [localhost]: FAILED! => {"msg": "the field 'args' has an invalid value, which appears to include a variable that is undefined. The error was: 'my_var' is undefined . . . snip . . .

PLAY RECAP
localhost      : ok=3    changed=0    unreachable=0    failed=1
```

Variable can also be defined in a separate file and referenced by the playbook. For example, this playbook first imports variables from a file that are play scoped, then imports a new set of variables within a task:

File: variable.yaml

```
---
- hosts: localhost
  vars_files:
  - vars/play_vars.yaml
  tasks:
  - debug:
      msg: "my_var is {{ my_var }}"

  - include_vars: file=task_vars.yaml
  - debug:
      msg: "my_var is {{ my_var }}"
```

```
$ cat vars/play_vars.yaml
my_var: "playvar"
$ cat vars/task_vars.yaml
my_var: "taskvar"
$ ansible-playbook variable.yaml
. . . snip . . .
TASK [debug]
ok: [localhost] => {
    "msg": "my_var is playvar"
}

TASK [include_vars]
ok: [localhost]

TASK [debug]
ok: [localhost] => {
    "msg": "my_var is taskvar"
}
. . . snip . . .
```

Variables - Inventory

host or group vars within inventory file
/etc/ansible/host_vars/*hostname*/*.yaml
/etc/ansible/group_vars/*group_name*/*.yaml

Defining Variables - Inventory

Inventory variables can be used to organize information by host or group. For example:

File: /etc/ansible/hosts

```
web1 role=master
web2 role=slave
web3 role=slave

[web]
web1
web2
web3

[web:vars]
poolid=42
```

File: variables_example.yaml

```
- name: configure master
  copy:
    src: "files/master-{{ poolid }}.conf
    dest: "/etc/web.conf"
  when: role == "master"

- name: configure slaves
  copy:
    src: "files/slave-{{ poolid }}.conf
    dest: "/etc/web.conf"
  when: role == "slave"
```

If the inventory has a lot of variables, it can become hard to read. Variables can be moved out of the file into a directory structure parallel to the inventory file. For example:

```
$ cat host_vars/web1/vars.yaml
role: master
$ cat group_vars/web/vars.yaml
poolid: 42
```

Variables - Registered

Result of any module can be captured
register: *varname*

Variables - Registered

Within a play, the output of any module can be captured into a variable. Most modules return a set of common values, and then module specific data. The data within the registered variable can then be manipulated using any of the Python or Jinja2 filters, and used within plays. For example:

File: register.yaml

```
---
- hosts: localhost
  tasks:
  - command: cat file
    register: output
  - debug: var=output
  - debug:
      msg: "You came in: {{ item|upper }}"
    with_items:
    - "{{ output.stdout_lines }}"
```

```
$ cat file
first
second
third
$ ansible-playbook register.yaml
TASK [debug]
ok: [localhost] => {
    "output": {
        "changed": true,
        ...snip...

        "stderr_lines": [],
        "stdout": "first\nsecond\nthird",
        "stdout_lines": [
            "first",
            "second",
            "third"
        ]
    }
}

TASK [debug]
ok: [localhost] => (item=first) => {
    "item": "first",
    "msg": "You came in: FIRST"
}
ok: [localhost] => (item=second) => {
    "item": "second",
    "msg": "You came in: SECOND"
}
ok: [localhost] => (item=third) => {
    "item": "third",
    "msg": "You came in: THIRD"
}
```

Facts

Gathered by executing code on managed hosts

- setup:
- *device_type*_facts; eg. ios_facts
- other programs supported: facter, ohai

Local facts /etc/ansible/facts.d/*

set_fact:

probar a ver los facts locales con:
ansible localhost -m setup

Defining Variables - Discovery

"Discovery", or "fact gathering" collects data about the host and assembles it into a complex variable that can then be used within plays. This is done using the setup module and when executed in a play, the output will show "TASK [Gathering Facts]". By default, this happens automatically at the start of every play, but this can be controlled via the config, or via the gather_facts: clause in the play itself.

The Puppet and Chef platforms also support fact gathering and if their respective gathering programs (facter, and ohai) are present on the remote host, then they are run and the resulting data is added to additional trees within the fact variable.

The setup module can gather a subset of facts, and can return just specific facts through the use of the filter option. See http://docs.ansible.com/ansible/latest/setup_module.html for details.

```
$ ansible localhost -m setup -a "filter=ansible_*_mb"
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_memfree_mb": 1227,
        "ansible_memory_mb": {
            "nocache": {
                "free": 3056,
                "used": 895
            },
            "real": {
                "free": 1227,
                "total": 3951,
                "used": 2724
            },
            "swap": {
                "cached": 0,
                "free": 511,
                "total": 511,
                "used": 0
            }
        },
        "ansible_memtotal_mb": 3951,
        "ansible_swapfree_mb": 511,
        "ansible_swaptotal_mb": 511
    },
    "changed": false,
    "failed": false
}
```

Examining Facts in a Playbook

The debug module can be used to examine facts from within a playbook; for example:

File: view_fact.yaml

```
---
- hosts: localhost
  tasks:
  - debug: var=ansible_hostname
```

```
$ ansible-playbook test.yaml
TASK [Gathering Facts]
ok: [localhost]

TASK [debug]
ok: [localhost] => {
    "ansible_hostname": "station1"
}
```

Facts can alternatively be saved into files for easy reference and examination. The following would save fact output to the facts/localhost file:

```
$ ansible localhost -m setup -t facts
```

Local Facts

Facts can be stored on individual managed hosts. If possible this is usually avoided as it effectively distributes the configuration across the hosts. Consider using the host_vars directory on the controller instead. Local facts are registered when the setup module runs, and placed into the ansible_facts subtree. All files found in /etc/ansible/facts.d/*.fact are processed, and can be either INI, JSON, or executable that emit JSON:

```
$ cat /etc/ansible/facts.d/rfc3092.fact
[metasyntactical_vars]
foo=bar
baz=qux
# cat /etc/ansible/facts.d/family.fact
["bryan", "sarah", "rebekah", "Julianne", "Brandon", "Lillian"]
# ansible localhost -m setup -a "filter=ansible_local"
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_local": {
            "family": [
                "Bryan",
                "Sarah",
                "Rebekah",
                "Julianne",
                "Brandon",
                "Lillian"
            ],
            "rfc3092": {
                "metasyntactical_vars": {
                    "baz": "qux",
                    "foo": "bar"
                }
            }
        }
    },
    "changed": false,
    "failed": false
}
```

Setting a Fact Within Play

New facts can be created on the fly within a play and be scoped to that run. For example:

File: set_fact.yaml

```
---
- hosts: localhost
  gather_facts: True
  tasks:
  - command: cat file
    changed_when: false
    register: result
  - set_fact:
      one: foo
      two: "{{ result.stdout_lines|count * 2 }}"
```

```
# cat file
foo
bar
# ansible-playbook facts.yaml -v
TASK [set_fact]
ok: [localhost] => {"ansible_facts": {"one": "foo",
       "two": "4"}, "changed": false, "failed": false}
```

DEMO: Facts

Registering a fact
Facts vs. variables evaluations

Setting Facts

estos playbooks están en /labfiles/solutions/chap4

Earlier we explored setting a fact based on an existing variable or static data. One powerful way to create new facts is using Ansible lookup functions. Let's try registering a fact based on the output of running a command:

File: ~/playbooks/facts_demo.yaml

```
+ ---
+ - hosts: localhost
+   gather_facts: false
+   tasks:
+   - set_fact:
+       fact_demo: "FACT: {{ lookup('pipe', 'date') }} "
+   - debug: var=fact_demo
```

```
$ ansible-playbook facts_demo.yaml
. . . snip . . .
TASK [set_fact]
ok: [localhost]

TASK [debug]
ok: [localhost] => {
    "fact_demo": "FACT: Thu Aug 31 15:01:16 MDT 2017 "
}
```

la función lookup con pipe conecta la salida del comando date a la cadena anterior en resumen genera la salida de date convertida a cadena

You have already seen examples where the set_fact module is used to store information about a host, or play. There is a subtle difference between variables and facts. Let's modify the play so that it outputs the fact more than once, with a pause between:

File: ~/playbooks/facts_demo.yaml

```
  . . . snip . . .
    - debug: var=fact_demo
+   - command: sleep 3
+   - debug: var=fact_demo
```

```
$ ansible-playbook facts_demo.yaml
. . . snip . . .
TASK [debug]
ok: [localhost] => {
    "fact_demo": "FACT: Thu Aug 31 15:06:56 MDT 2017 "
}

TASK [command]
changed: [localhost]

TASK [debug]
ok: [localhost] => {
    "fact_demo": "FACT: Thu Aug 31 15:06:56 MDT 2017 "
}
```

Note how the variable referencing the fact has the same value both times. It is only evaluated when the fact is first set. Ansible does not re-evaluate facts after they have been set!

Var vs. Fact Evaluation Behavior

Now let's try a similar test using a var:

File: vars_demo.yaml

```
+ ---
+ - hosts: localhost
+   gather_facts: false
+   vars:
+     var_demo: "VAR: {{ lookup('pipe', 'date') }} "
+   tasks:
+   - debug: var=var_demo
+   - command: sleep 3
+   - debug: var=var_demo
```

```
$ ansible-playbook facts_demo.yaml
. . . snip . . .
TASK [debug]
ok: [localhost] => {
    "var_demo": "VAR: Thu Aug 31 15:13:42 MDT 2017 "
}

TASK [command]
changed: [localhost]

TASK [debug]
ok: [localhost] => {
    "var_demo": "VAR: Thu Aug 31 15:13:45 MDT 2017 "
}
```

Surprise! Vars are evaluated every time they are referenced.

Inclusions

Make playbooks more

- readable
- modular

Playbooks, Tasks, and vars can be imported

`include*` **vs.** `import*` **differences complex**

- when in doubt, `import`

Inclusions

As playbooks get larger and more complex it can be useful to re-factor them into smaller logical units and then use import, or include statements to tie those together into a more readable playbook. This also allows the individual, more targeted plays, to be potentially be reused elsewhere. A simple alternative might be careful use of tags to allow parts of the playbook to be used in isolation. Roles, covered later, are an even more comprehensive solution that allows an entire collection of related Ansible playbook assets to be packaged and referenced.

The following example show the use of inclusions shows how a complex playbook might be assembled by importing task list from other files:

File: site.yaml

```
---
- hosts: app1
  tasks:
  - import_tasks: install.yaml
  - import_tasks: configure.yaml
  - import_tasks: init.yaml
```

The individual task files would then be structured as a list of bare task; for example:

File: tasks/install.yaml

```
- name: install base web
  yum: name=httpd state=latest
- name: grab app config vars
  include_vars: vars/app1.yaml
- name: install app1
  git:
    repo: "https://code.example.org/app1.git"
    dest: "{{ target_dir }}"
    version: "{{ release }}"
```

Ansible has two modes of operation for reusable content: dynamic and static. The full difference between import (static), and include (dynamic) is somewhat complex and has changed a couple times within the 2.X release series. Consult the docs for your specific version and test playbook operation carefully.

Briefly, Ansible pre-processes all static imports during Playbook parsing time. Dynamic includes are processed during runtime at the point in which that task is encountered, allowing them to be constructed from values such as facts or other registered data.

la diferencia entre inlude e import es que import lo hace al principio de la ejecución del playbook por lo tanto se considera estático, mientras que include lo hace durante la ejecución en el punto del código donde se encuentra y por lo tanto es dinámico y puede usar facts que hayan sido modificados en ese momento

Lab 4

Estimated Time:
R7: 45 minutes

Task 1: Variables and Facts [R7]

Page: 4-14 Time: 20 minutes

Requirements: 💻 (1 station)

Task 2: Inclusions [R7]

Page: 4-22 Time: 25 minutes

Requirements: 💻 (1 station)

Objectives

- Use variables and understand variable precedence
- Gather and use facts

Requirements

- (1 station)

Relevance

Lab 4
Task 1
Variables and Facts [R7]

Estimated Time: 20 minutes

estos playbooks están en /labfiles/solutions/chap4.
copiarlos a un directorio local para poderlos ejecutar y modificar

1) Run the Ansible setup module to view gathered facts for localhost:

```
$ ansible localhost -m setup
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_all_ipv4_addresses": [
            "172.17.0.1",
            "10.100.0.4"
]
. . . snip . . .
```

2) Create a file called facts_test.yaml with the following contents:

File: facts_test.yaml

```
+ ---
+ - hosts: localhost
+   tasks:
+   - debug: msg="Hostname is {{ ansible_fqdn }}"
+   - debug: msg="IP list contains {{ ansible_all_ipv4_addresses }}"
```

3) Run the facts_test.yaml file as a play:

```
$ ansible-playbook facts_test.yaml
PLAY [localhost]

TASK [Gathering Facts]
ok: [localhost]

TASK [debug]
ok: [localhost] => {
    "msg": "Hostname is stationX.example.com"
```

```
}

TASK [debug]
ok: [localhost] => {
    "msg": "IP list contains [u'172.17.0.1', u'10.100.0.X']"
}
PLAY RECAP
localhost     : ok=3    changed=0    unreachable=0    failed=0
```

Note that the IP information is stored as a list object. The u'string' notation here is the way python marks a string as unicode rather than a byte string. The result is this list of two separate unicode strings, each representing an IP address discovered while gathering facts.

Defining variables before a play

4) Create a file called `var_test.yaml` with the following contents:

File: var_test.yaml

```
+ ---
+ - hosts: localhost
+   vars:
+     myvar1: "Hello"
+     myvar2: "World"
+   tasks:
+   - debug: msg="Your message is {{ myvar1 }}, {{ myvar2 }}"
```

5) Run the playbook to view the output:

```
$ ansible-playbook var_test.yaml
OUTPUT:
PLAY [localhost]

TASK [Gathering Facts]
ok: [localhost]

TASK [debug]
ok: [localhost] => {
    "msg": "Your message is Hello, World"
}

PLAY RECAP
```

```
localhost     : ok=2    changed=0    unreachable=0    failed=0
```

Observe the output from the playbook. TASK "Gathering Facts" was run even though we aren't using any facts in this playbook.

6) Disable fact gathering in your playbook by:

File: var_test.yaml

```
   ---
   - hosts: localhost
+    gather_facts: false
     vars:
       myvar1: "Hello"
       myvar2: "World"
     tasks:
     - debug: msg="Your message is {{ myvar1 }}, {{ myvar2 }}"
```

7) Run the playbook again to see the difference in output:

```
$ ansible-playbook var_test.yaml
PLAY [localhost]

TASK [debug]
ok: [localhost] => {
"msg": "Your message is Hello, World"
}

PLAY RECAP
localhost     : ok=1    changed=0    unreachable=0    failed=0
```

The "Gathering facts" task has been skipped.

8) Run the playbook again, this time overriding the variable value for `myvar2` by passing a command-line option.

```
$ ansible-playbook var_test.yaml --extra-vars "myvar2=Ansible"
. . . snip . . .
TASK [debug]
ok: [localhost] => {
"msg": "Your message is Hello, Ansible"
```

```
}
```

Fact Gathering Dependencies

9) Get a rough profile of how much data is currently contained within the fact variable:

```
$ ansible localhost -m setup > /tmp/basefacts
$ ansible localhost -m setup -b > /tmp/basefacts2
$ wc /tmp/basefacts*
  973  1888 39273 /tmp/basefacts
 1042  1978 41207 /tmp/basefacts2
 2015  3866 80480 total
```

Haciendo este paso es suficiente para ver que con privilegios escalados se ven más facts que sin ellos

Why are the two files different sizes?

10) Compare fact gathering for specific subtrees with and without the become option:

NO ES NECESARIO PROBAR EL RESTO DE ESTOS PASOS

```
$ ansible localhost -m setup -a "filter=ansible_system_capabilities"
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_system_capabilities": [
            ""
        ]
    },
    "changed": false,
    "failed": false
}
$ ansible localhost -m setup -a "filter=ansible_system_capabilities" -b
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_system_capabilities": [
            "cap_chown",
            "cap_dac_override",
            "cap_dac_read_search",
. . . snip . . .
            "36+ep"
        ]
    },
    "changed": false,
    "failed": false
```

```
}
$ ansible localhost -m setup -a "filter=ansible_lvm"
. . . output omitted . . .
$ ansible localhost -m setup -a "filter=ansible_lvm" -b
. . . output omitted . . .
```

Conclusion: certain facts require root access to gather.

11) Try gathering network facts with and without the ip program being present:

NO ES NECESARIO EJECUTAR
DEMUESTRA QUE CIERTOS FACTS
REQUIEREN QUE DETERMINADOS EJECUTABLES
ESTÉN PRESENTES

```
$ ansible localhost -m setup -a "filter=ansible_eth*"
localhost | SUCCESS => {
    "ansible_facts": {
        "ansible_eth0": {
            "active": true,
            "device": "eth0",
            "features": {
                "busy_poll": "off [fixed]",
                "fcoe_mtu": "off [fixed]",
. . . snip . . .
$ sudo mv /sbin/ip /sbin/ip.disabled
$ ansible localhost -m setup -a "filter=ansible_eth*"
localhost | SUCCESS => {
    "ansible_facts": {},
    "changed": false,
    "failed": false
}
$ sudo mv /sbin/ip.disabled /sbin/ip
```

Conclusion: certain facts require specific system binaries to be present for gathering.

Extending Fact Gathering

Esta parte es opcional porque usa el fact checker de CHEF

12) Install the Chef client which contains the ohai program and see how it effects the fact gathering:

```
$ sudo yum install -y /labfiles/chef-*rpm
. . . output omitted . . .
$ ansible localhost -m setup -b > /tmp/extended_facts
$ wc /tmp/basefacts2 /tmp/extended_facts
1317   2497  52002 /tmp/basefacts2
```

```
16536  29536 571892 /tmp/extended_facts
```

13) Examine a specific subtree produced by the ohai module:

```
$ ansible localhost -m setup -a "filter=ohai_languages"
localhost | SUCCESS => {
    "ansible_facts": {
        "ohai_languages": {
. . . snip . . .
                "archname": "x86_64-linux-thread-multi",
                "version": "5.16.3"
            },
            "python": {
                "builddate": "Aug 2 2016, 04:20:16",
                "version": "2.7.5"
            },
            "ruby": {}
        }
    },
    "changed": false,
    "failed": false
```

14) Create your own script that extends the facts to include a full list of installed packages along with their size:

Esta parte es interesante para ver como generar facts

File: /tmp/packages.fact

```
+ #!/bin/sh
+ echo -n {
+ rpm -qa --queryformat '"%{NAME}":"%{SIZE}",' | sed 's/.$//'
+ echo -n }
```

```
$ sudo mkdir /etc/ansible/facts.d/
$ sudo cp /tmp/packages.fact /etc/ansible/facts.d/
$ sudo chmod +x /etc/ansible/facts.d/packages.fact
```

ESTE EJEMPLO CREA UN SCRIPT QUE RECOPILA EL SOFTWARE INSTALADO Y LO CONVIERTE EN FACTS.

POSTERIORMENTE SE GENERA UN REPORT QUE USA ESOS FACTS PARA GENERAR UN LISTADO DE SOFTWARE CON UN TAMAÑO SUPERIOR A UNA CANTIDAD

15) Verify your new facts are working:

```
$ ansible localhost -m setup -a "filter=ansible_local" | head
localhost | SUCCESS => {
    "ansible_facts": {
```

```
        "ansible_local": {
            "packages": {
                "GConf2": "6605076",
                "GeoIP": "2905020",
                "ModemManager-glib": "1059328",
                "NetworkManager": "10605587",
                "NetworkManager-config-server": "1292",
                "NetworkManager-glib": "939060",
```

16) Create a simple play that uses your custom facts to generate a report with the names of all packages larger than a specific size:

File: ~/playbooks/report.yaml

```
+ ---
+ - hosts: localhost
+   tasks:
+   - name: Generate report of installed large packages
+     shell: "echo {{ item.key }} >> /tmp/report"
+     with_dict: "{{ ansible_local.packages }}"
+     when: item.value|int > 15000000
```

17) Execute the play and review the report:

```
$ ansible-playbook report.yaml
TASK [Gathering Facts]
ok: [localhost]

TASK [Generate report of installed large packages]
skipping: [localhost] => (item={'key': u'imsettings-libs', 'value': u'532479'})
skipping: [localhost] => (item={'key': u'kbd-misc', 'value': u'2366207'})
skipping: [localhost] => (item={'key': u'perl-Thread-Queue', 'value': u'27642'})
skipping: [localhost] => (item={'key': u'pth', 'value': u'267851'})
skipping: [localhost] => (item={'key': u'perl-HTML-Tagset', 'value': u'19784'})
skipping: [localhost] => (item={'key': u'iputils', 'value': u'339417'})
changed: [localhost] => (item={'key': u'wqy-zenhei-fonts', 'value': u'17156598'})
. . . snip . . .
$ cat /tmp/report
wqy-zenhei-fonts
libpurple
linux-firmware
```

```
ghostscript
. . . snip . . .
```

Cleanup

18) Remove the Chef client software, and custom facts:

```
$ sudo yum remove -y chef
$ sudo rm -f /etc/ansible/facts.d/*
```

Objectives

- Assemble a playbook using includes
- Use variables in various scopes

Requirements

- (1 station)

Relevance

Lab 4
Task 2
Inclusions [R7]

Estimated Time: 25 minutes

1) When using inclusions, it is important to organize content. Create directories that can hold the task and var files for this exercise:

```
$ mkdir -p ~/playbooks/includes/{vars,tasks}
$ cd ~/playbooks/includes
```

EN LUGAR DE PLAYBOOKS, COMO SE HACE CON CONTENEDORES
ES MÁS CÓMODO HACER LOS DIRECTORIOS EN /home/guru/docker
$ mkdir -p /home/guru/docker/{vars,tasks}
$ cd /home/guru/docker/
Copiar en cada directorio los siguientes playbooks que están en /labfiles/solutions/chap4

2) Create a generic list of tasks that can be used to install and start a service based on passed variables:

File: tasks/install.yaml — cp /labfiles/solutions/chap4/install.yaml tasks/

```
+ ---
+ - name: Install {{ service }}
+   yum: "name={{ service }} state=installed"
+
+ - name: Enable and Start {{ service }}
+   service: "name={{ service }} enabled=yes state=started"
```

3) Create another list of tasks that can configure a simple firewall and open a port based on passed variables:

File: tasks/firewall.yaml cp /labfiles/solutions/chap4/firewall.yaml tasks/

```
+ ---
+ - name: Open TCP port {{ port }}
+   firewalld:
+     port: "{{ port ~ '/tcp'}}"
+     immediate: true
+     permanent: true
+     state: enabled

+ - firewalld:
+     interface: eth0
+     immediate: true
+     permanent: true
+     state: enabled
+     zone: public
```

4) Create a master playbook that references your tasks via includes, and also pulls in additional variables:

File: ~/playbooks/includes/site.yaml cp /labfiles/solutions/chap4/site.yaml.v1 ./site.yaml

```
+ ---
+ - hosts: web
+   vars:
+     port: 8000
+   tasks:
+   - name: Install web
+     import_tasks: tasks/install.yaml
+     vars:
+       service: httpd

+   - name: Install firewall
+     import_tasks: tasks/install.yaml
+     vars:
+       service: firewalld

+   - name: Configure Firewall
+     import_tasks: tasks/firewall.yaml
```

LAS TAREAS INCLUIDAS EN install.yaml AL SER IMPORTADAS DOS VECES SE REPITEN CADA UNA DE ELLAS VARIANDO EL SERVICIO QUE SE INSTALA Y ACTIVA EN CADA CASO

5) Test your playbook by creating a base container and then executing the playbook against it:

```
$ cd ~/docker
$ sh  /labfiles/docker_centos_init.sh web
$ ansible-playbook ~/playbooks/includes/site.yaml
PLAY [web]

TASK [Gathering Facts]
ok: [web]

TASK [Install httpd]
changed: [web]

TASK [Enable and Start httpd]
changed: [web]

TASK [Install firewalld]
changed: [web]

TASK [Enable and Start firewalld]
changed: [web]

TASK [Open TCP port 8000]
changed: [web]

TASK [firewalld]
ok: [web]

PLAY RECAP
web        : ok=7    changed=6    unreachable=0    failed=0
```

6) Modify the playbook so that it has a new first task that pulls in a file with variables:

File: ~/playbooks/includes/site.yaml agregar estas lineas a site.yaml.v1

```
    tasks:
+   - name: Import content
+     include_vars: vars/content.yaml

    - name: Install web
```

7) Create the referenced variable file:

cp /labfiles/solutions/chap4/content.yaml vars/

File: ~/playbooks/includes/vars/content.yaml

```
+ ---
+ web_content: "Welcome to the {{ ansible_fqdn }} website!"
```

8) Modify your site.yaml adding three new tasks, after those already defined, that do the following:

cp /labfiles/solutions/chap4/site.yaml.v2 site.yaml

1. Configure the Apache service to listen on the port listed by the port variable.
   - Use lineinfile module
   - modify the /etc/httpd/conf/httpd.conf file
   - change the Listen directive
2. Create a default page for the website
   - Use the copy module
   - create the /var/www/html/index.html file
   - file must contain the result of processing the Jinja of the included web_content var
3. Restart the web service.

9) Test your playbook changes. The result should match the following output:

```
$ ansible-playbook ~/playbooks/includes/site.yaml
. . . snip . . .
TASK [Configure web for alternate port]
changed: [web]

TASK [Create index.html]
changed: [web]

TASK [Restart web]
changed: [web]

PLAY RECAP
web        : ok=11   changed=5    unreachable=0    failed=0
```

10) Determine the IP of the container using data from the dynamic inventory script and then verify it is serving the expected content:

```
$ ./docker.py --host web | tr ',' '\n' | grep -m1 IPAddress
172.18.0.2
$ curl 172.18.0.2:8000
 Welcome to the 94f402b8a61d website!
```

11) Modify the value of the port variable and verify that the playbook works as expected to reconfigure the site:

File: site.yaml

```
   ---
   - hosts: web
     vars:
→      port: 80009000
```

```
$ ansible-playbook ~/playbooks/includes/site.yaml
. . . snip . . .
ASK [Open TCP port 9000]
changed: [web]

TASK [firewalld]
ok: [web]

TASK [Configure web for alternate port]
changed: [web]

TASK [Create index.html]
ok: [web]

TASK [Restart web]
changed: [web]

PLAY RECAP
web        : ok=11   changed=4    unreachable=0    failed=0
```

Cleanup

12) Remove the web container:

```
$ docker container rm -f web
```

Content

Chapter 5

JINJA2 TEMPLATES

Templating Overview

Jinja2 is a general purpose templating language. It is a library for Python designed to be flexible, fast and secure. If you've had any experience with other templating engines such as Smarty for PHP, or Django, the syntax will be familiar. Ansible greatly expands the number of filters and tests available, as well as adding a new Jinja2 plugin type called "lookups".

Jinja2 is used throughout Ansible playbooks, especially when referencing variables in a string; for example:

```
"The value of my_variable is {{ my_variable }}"
```

Jinja2 can also express logical operations such as loops and conditions. For example:

```
{# Process app list #}
{% for app in myapps if not app == "xiphias" %}
Processing app number {{ loop.index }} - {{ app }}
{% endfor %}
```

It is important to note that templating and Jinja2 processing happens on the Ansible controller side, not on the target hosts, so all the information used in a playbook will be sourced from the controller.

The following example shows that a "Template" is a type of object that contains a string with some special formatting tags "{{ }} or {% %}". Jinja2 takes the string and parses out the special characters to perform dynamic replacements.

Jinja2

General purpose template language

- Python library

Delimiters

- {% *expression* %} → logic
- {{ *expression* }} → output results
- {# *expression* #} → comment

Whitespace Control

- trim_blocks (enabled) → first newline after a template tag is removed automatically
- lstrip_blocks (disabled) → does not strip tabs and spaces from the beginning of a line to the start of a block
- Manually toggle lstrip_blocks per statement my adding minus of plus sign to delimiter

File: jinja2_example.py

```
from jinja2 import Template
t = Template("Hello {{ some_variable }}")
print t.render(some_variable="World")
print t.render(some_variable="!NON ROBOTS!")
t2 = Template("Here are some numbers:
      {% for i in range(1,12) %}{{ i }} " "{% endfor %}")
print t2.render()
```

```
Hello World
Hello !NON ROBOTS!
Here are some numbers: 1 2 3 4 5 6 7 8 9 10 11
```

Ansible works by using the exact same engine. Each object (string, integer, boolean, etc.) we use in a play is passed as a template to Jinja2. If Jinja2 finds its special characters, it will perform a replacement. We can see Jinja2 parse our loop by running a task such as:

File: jinja_example.yaml

```
- debug: msg="Here are some numbers
       {% for i in range(1,12) %}{{i}} {% endfor %}"
```

```
TASK [debug]
ok: [localhost] => {
   "msg": "Here are some numbers 1 2 3 4 5 6 7 8 9 10 11 "
}
```

Whitespace Control

Jinja2 as called by Ansible tries to "do the right thing" and strip only whitespace introduced by the delimited expressions. All other whitespace is treated as literal and rendered through. If you need to control this behavior, you can configure the `lstrip_blocks` and `trim_blocks` settings by adding a line at the top of your template file. For example, compare the default output of this simple template with and without the config line:

File: `template_demo.j2`

```
{% for name in ['Bryan', 'Dax'] %}
    {{ name }}
{% endfor %}
```

```
    Bryan
    Dax
```

Would render to:

File: `whitespace_demo1.j2`

```
+ #jinja2: lstrip_blocks: "True"
  {% for name in ['Bryan', 'Dax'] %}
      {{ name }}
  {% endfor %}
```

```
Bryan
Dax
```

If you need to control this behavior on a per statement basis, then append the plus and minus symbols to the delimiter tag. For example (note that the "num" var has 2 spaces on each side of it in the template file):

File: `whitespace_demo2.j2`

```
{% for num in seq %}
  {{ num }}
{% endfor %}

{% for num in seq -%}
  {{ num }}
{% endfor %}

{% for num in seq %}
  {{ num }}
{%- endfor %}

{% for num in seq -%}
  {{ num }}
{%- endfor %}
```

```
# cat -A whitespace_demo2.j2.output
  1  $
  2  $
  3  $
  4  $
$
1  $
2  $
3  $
4  $
$
  1  2  3  4$
1234$
```

If the evaluation of the variable itself is producing the unwanted whitespace, then use a filter like `trim()` to process the data instead.

Expressions

Evaluated and result printed
Evaluation depends on data type
Methods exist to convert between data-types
Operations

- Math
- Comparisons
- other...

Expressions

The most commonly seen Jinja2 block in Ansible is the "expression" block. Expressions can be used to simply print the value of a variable, but can also perform comparisons, math, and filter data. When performing operations within an expression, it is important to understand the lierals, or data-types, available. Trying to perform operation on the wrong type can result in errors or surprising results. The following explores the types available:

String

Example: `"Hello World"`

Everything between the double quotes is a string.

Integer

Example: `42`

An integer is a number with a precision of 0, or nothing after the decimal place. NOTE: "42" in quotes, is actually considered a string. This prevents Python from being able to apply integer-specific operations such as some math.

Float

Example: `42.42`

If your number has a decimal place at all (like 42.0), Python (and therefore Ansible) will consider this a "float" value. This is important when performing comparisons and certain math operations as the result.

List

Example: `['list','of','objects',42]`

A list is a collection of objects, or other "literals". These can be of combined types such as a list of strings and integers. A common use for lists is to be used during iteration during a "for" or "while" loop.

Tuple

Example: `('tuples','resemble','lists')`

A tuple is a "list" that can not be modified after its creation. It is marked as immutable and lacks the ability to add/remove items at the object attribute level. This means you won't find a Tuple.append("item"), when there is a List.append("item").

Dictionary or "dict"

Example: `{'key':'value', 'key2':'value2'}`

A "dict" is a Python structure that combines keys and values. Keys are always unique and have one value. These are less commonly seen in templates, but Ansible populates some "dict" objects during Fact Gathering.

Expression - Math

Caution: "Math" operations in Python can surprise you! Python's "add"

function, usually represented by "+", detects the input object types and performs different functions based on the result. Adding two strings (concatenate):

```
>>> "String" + "String2"
'StringString2'
```

Adding a string and integer:

```
>>> "String" + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

This fails because a string can't be added to an integer.

Adding two strings that look like they should be integers:

```
>>> "2"+"2"
'22'
```

This just concatenates the two strings, which may not be the desired result!

And finally, adding two integers as expected:

```
>>> 2+2
4
```

Math Operators Supported

Operator	Description
+	Addition
-	Subtract
/	Divide (Returns float)
//	Divide (Returns integer)
%	Calculate remainder (modulus)
*	Multiply
**	Raise left operand to the power of right operand. {{ 2**3 }} results in 8.

Expressions - Comparisons

Comparison	Description
+	Addition
==	Checks equality
!=	Checks for inequality
>	true if left number is greater than right number
>=	true if greater than or equal to
<	less than
<=	less than or equal to

The most common use of Jinja2 comparison operators in Ansible is within the context of `when:` clauses, but they can be used within templates:

```
$ ansible localhost -e letter="A"↵
      -m debug -a "msg={{ letter == 'A' }}"
localhost | SUCCESS => {
  "msg": true
}
$ ansible localhost -e '{"guess":5}' -m debug↵
      -a 'msg="{% if guess != 5 %}Try again{% endif %}"'
localhost | SUCCESS => {
    "msg": ""
}
```

Logic Operators

Many types of expressions can be combined together using logical operators. The following four logical operators are supported:

Operator	Description
and	return true if the left and the right operand are true
or	return true if the left or the right operand are true
(expr)	group and expression
not	negates a statement

QUIZ: Jinja2 Templates

Quiz and group discussion

1. Which of the following delimiters is used to include logic operations such as if/then?
   - {% expr %}
   - {{ expr }}
   - {@ expr @}
   - {# expr #}
2. What would be the result of processing this Jinja2 template: {{ for }}
   - The literal string "for" would be printed.
   - Would result in a type error.
   - Would begin a Jinja2 for-loop.
   - The value of the variable named "for" would be printed.
3. What would be the result of processing this Jinja2 template: {{ "7"+"11" }}
   - Would result in a type error.
   - The literal string "711" would be printed.
   - The sum of the variables named "7" and "11" would be printed.
   - The result "18" would be printed.
4. What would be the result of processing this Jinja2 template: {{ (7 < 9) and (5 != 4) }}
   - Would return the string "true"
   - Would return the boolean value "true"
   - Would return the string "false"
   - Would return the boolean value "false"
5. Assuming vars "nope: false", and "yep: true", which of the following templates would result in: "boom"?
   - {% if not nope %}boom{% endif %}
   - {% if yep and not nope %}boom{% endif %}
   - {% if nope is defined%}boom{% endif %}
   - {% if yep is not defined%}boom{% endif %}
   - {% if nope is undefined%}boom{% endif %}
6. Which of the following delimiters is used to output the rendered results?

- {% expr %}
- {{ expr }}
- {@ expr @}
- {# expr #}

Filters

Filters
- Jinja2 native
- Ansible extensions

Transform and replace
Default values and "omit"
Manipulate lists
Callable functions (generate data)

Expressions - Other Operators

Besides the common operators, Jinja2 supports a few special operators that are very useful, but don't seem to fit any other category.

Operator	Description
in	Test to see if an object appears in a set.
is	Perform a test.
|	Applies a filter.
~	Convert all operands into strings and concatenate.
()	Call a method, or callable attribute.
. and []	Get an attribute of an object.

Transforming Data with Filters

Often, the data available is not quite in the right structure to be used, or is required in a different form on output. Jinja provides many filters that can transform data in a variety of useful ways. Ansible extends these filters adding more that are particularly useful within the context of playbooks, and manipulating data commonly found on the systems it manages. Filters can be used to transform data inside any template expression. Consider the following examples:

```
{{ variable | to_json }}
{{ variable | to_nice_yaml(indent=8)}}
{{ variable | capitalize | center(width=80) }}
{{ listvar | sort }}
```

Replacements

Jinja provides two filters for replacing one pattern of data with another. For simple operations, the "replace" filter can be used and will return a copy of the value with all occurrences of a substring replaced with a new one. The first argument is the substring that should be replaced, the second is the replacement string. If the optional third argument count is given, only the first count occurrences are replaced; for example:

```
{{ "Chef is the best!"|replace("Chef", "Ansible") }}
   → Ansible is the best!
{{ "54321..."|replace(".", "Boom", 2) }}
   → 54321BoomBoom.
```

For more complicated replacements, regular expresssions can be used including using references to captured data; for example:

```
{{ 'foobar' | regex_replace('^f.*o(.*)$', '\\1') }}
   → bar
```

Ansible Filter Extensions

An example of an additional filter added by Ansible is the ipaddr filter which allows several IP address queries to be performed on a string, or set. This allows for dynamically detecting valid IPs in a range, or

subnet. If the filter matches, it will output the resulting IP address; for example:

```
{{ '192.168.0.1' | ipaddr }}
   → 192.168.0.1

{{ '192.268.0.1' | ipaddr }}
   → false
```

This filter is extremely powerful! See http://docs.ansible.com/ansible/latest/playbooks_filters_ipaddr.html#basic-tests for more details.

Default Values

Generally variables have a value set and then filters might manipulate that value is some way. The default filter can be used when you want to provide a value when a variable is undefined; for example:

```
{{ number | default(42) }}
```

Since Ansible 1.8, it is possible to use the default filter to omit module parameters using a special variable called "omit". This can only be done on module arguments that are not required, and will cause the module to use its internal default value; for example:

File: example.yaml

```
- name: Create empty files
  file: >
    dest={{item.path}} state=touch
    mode={{item.mode|default(omit)}}
  with_items:
    - path: /tmp/foo
      mode: "0640"
    - path: /tmp/bar
    - path: /tmp/baz
      mode: "0444"
```

Two of the files in the above example have a mode listed, and therefore will be passed as an argument to the file module. Rather than asking Ansible to use a default value for the mode on /tmp/bar, it will be omitted from the command entirely, and use the user's umask to determine initial permission.

Filters for Lists

There are many filters that can be applied to lists of objects. The following shows identifying the minimum and maximum values, returning a single random element, and shuffling the list:

```
mylist: [3,4,2,3,7]

{{ mylist | max }}
   → 7

{{ mylist | min }}
   → 2

{{ mylist | random }}
   → 3

{{ mylist | shuffle }}
   → 2, 4, 3, 7, 3  # Warning: random, i.e. non-idempotent

{{ mylist | shuffle(seed=inventory_hostname) }}
   → 7, 3, 4, 2, 3  # Warning: idempotent, i.e non-random
```

The random filter can also be used to generate a random number based on a range; for example:

```
{{ 50 | random }} {# random number from 0-50 #}
   → 42

{{ 100 | random(start=1, step=2) }} {# random odd between 1-99 #}
   → 67
```

Set Theory Filters

```
list1: [1,2,3,3,4,5]
list2: [4,5,6,7,8]

{{ list1 | unique }}
   → 1, 2, 3, 4, 5

{{ list1 | union(list2) }}
   → 1, 2, 3, 4, 5, 6, 7, 8

{{ list1 | intersect(list2) }}
   → 4, 5

{{ list1 | difference(list2) }}
   → 1, 2, 3
```

```
{{ list1 | symmetric_difference(list2) }}
   → 1, 2, 3, 6, 7, 8
```

Callable Functions

Jinja2 has a handful of globally callable functions already defined, and can easily be extended. Functions often are used to produce data instead of manipulating it. The following examples show using functions to generate a sequence, and some filler text:

```
{% for i in range(10,50,5) %}user{{i}}, {% endfor %}
   → user10, user15, user20, user25, user30, user35, user40, user45,
```

```
{{ lipsum(html=true,n=3,min=10,max=80) }}
   → <p>Senectus eros nostra mi bibendum iaculis, conubia sit
   . . . snip . . .
```

Tests

Tests
- is
- in
- match vs search

version_compare
Path tests
Task result tests

Tests

Tests are essentially a filter that returns either true or false (boolean). Tests can be useful when validating an expression, such as in a "when" clause.

Jinja2 Reference:
http://jinja.pocoo.org/docs/2.9/templates/#builtin-tests

Ansible Docs:
http://docs.ansible.com/ansible/latest/playbooks_tests.html

The simply way to perform a test is by using the "is" operator. To use a test with "is", the syntax looks like: `{{ var is test }}` where *test* gets replaced with the name of a test; for example:

```
$ ansible localhost -m debug -a "msg={{ 1 is string }}"
localhost | SUCCESS => {
    "msg": false
}
$ ansible localhost -m debug -a "msg={{ 'one' is string }}"
localhost | SUCCESS => {
    "msg": true
}
```

In this first example, 1 is the object, "is" tells me I am performing a test, and that test is called "string". The second example shows the same test being performed on a string object. Common tests using this syntax include: defined, undefined, divisibleby(n), odd, even, none, number, and string. A complete reference can be found in the Jinja2 official documentation:
http://jinja.pocoo.org/docs/2.9/templates/#builtin-tests

The other test operator is the "in" operator which returns true if the provided value is a member of the provided set. The following examples show usage:

```
{{ "one" in ['one','two','three'] }}
   → true

{% if host not in groups['web'] %} ... {% endif %}

{% if (var1 is defined) and (var1 is number) and (var1 is odd)%}
  What an odd number!
{% else %}
  var not defined
{% endif %}
```

Match and Search Tests

The Jinja match and search tests can be used to determine if a pattern matches another string. When using the built-in test "match", the regular expression must match the entire string. Using "search" will allow for substring matches (much like how grep works):

File: search_example.yaml

```
vars:
  myvar: "This is my string"

tasks:
  - debug: msg="The value of myvar contains string"
    when: myvar | search("string")
```

```
TASK [debug]
ok: [localhost] => {
"msg": "The value of myvar contains the word string"
}
```

Version Comparison

One type of string that often need to be tested against, is a software version string. Ansible provides a special test that is optimized for the operation. For example, the following exits if the playbook executes on a host with a Linux distro version less that that given in the var:

File: version_compare.yaml

```
vars:
 var: '17.10'
tasks:
- assert:
    msg: "Ansible version
       {{ansible_distribution_version}} is less than {{var}}"
    that:
    - ansible_distribution_version | version_compare(var, 'ge')
```

Common Path Tests

Path tests used in a `when:` clause provide an efficient way to execute a task based on certain file properties. This works without having to invoke the `stat` or similar module, registering the results, and then testing; for example:

File: path_tests.yaml

```
- debug: msg="path is a directory"
  when: mypath|is_dir

- debug: msg="path is a file"
  when: mypath|is_file

- debug: msg="path is a symlink"
  when: mypath|is_link

- debug: msg="path already exists"
  when: mypath|exists
```

Task Results as a Test

The standard object returned by task execution can easily be checked. This can be used to execute tasks based on the exit results of a previous task; for example:

File: tast_results_test.yaml

```
tasks:

  - shell: /usr/bin/foo
    register: result
    ignore_errors: True

  - debug: msg="it failed"
    when: result|failed

  - debug: msg="it changed"
    when: result|changed

  - debug: msg="it succeeded in Ansible >= 2.1"
    when: result|succeeded

  - debug: msg="it succeeded"
    when: result|success

  - debug: msg="it was skipped"
    when: result|skipped
```

Lookups

Data from external sources

- execute on local system
- cwd relative to role or play

Lookup modules

- CSV
- INI
- pipe
- *many* more

Lookups

The Lookup module is considered an advanced feature of Ansible. The lookup plugins are designed to extract data from other sources such as a database, CSV, or INI file. For example, getting the contents of a text file:

File: lookup_text.yaml

```
---
- hosts: localhost
  vars:
    file1: "{{ lookup('file', 'myfile.txt') }}"
  tasks:
  - debug: msg="Your file contains {{ file1 }}"
```

```
$ ansible-playbook file_lookup.yaml
TASK [debug]
ok: [localhost] => {
    "msg": "Your file contains This is some data\nIt came from a 2-line file"
}
```

Parsing through an INI or CSV file

File: sample.ini

```
[section1]
user=section1_user

[section2]
user=section2_user

[section3]
user=section3_user
```

File: ini_lookup.yaml

```
---
- hosts: localhost
  tasks:
  - debug: msg="Section 1 User is {{ lookup('ini', 'user section=section1 file=sample.ini') }}"
  - debug: msg="Section 2 User is {{ lookup('ini', 'user section=section2 file=sample.ini') }}"
```

```
$ ansible-playbook ini_lookup.yaml
TASK [debug]
ok: [localhost] => {
   "msg": "Section 1 User is section1_user"
}

TASK [debug]
ok: [localhost] => {
    "msg": "Section 2 User is section2_user"
}
```

Using Live Data from a Pipe

```
$ ansible localhost -m debug -a "msg={{lookup('pipe', 'date')}}"
localhost | SUCCESS => {
    "msg": "Tue Aug 29 15:34:40 MDT 2017"
}
```

The documentation has a hard time keeping up with the lookup modules. It is best to check the Ansible source repo on GitHub to see the full list.

Control Structures

for → **loop over items in list**
if → **test and branch**

Loops and Conditional Branching

Looping over a set of data, and taking different actions based on tests on data are often needed in templates. Jinja2 uses the {% for ... %} and {% endfor %} statements to define a block that loops over an iterable set of data. Conditional execution of code can be handled by the {% if ... %}, {% elif ... %}, {% else %}, {% endif %} construct. The following examples use this set of data and show the use of simple for and if statements:

File: vars

```
---
list: [1,2,3]
hash: {"a":1, "b":2, "c":3}
```

File: simple_control_1.j2

```
Items from list:
{% for item in list %}
{{ item }}
{% endfor %}
```

```
Items from list:
1
2
3
```

Hashes are more complex to process depending on whether you need the key names, values, or both:

File: simple_control_2.j2

```
Keys from hash:
{% for key in hash %}
 {{ key }}
{%- endfor %}

Values from hash:
{% for value in hash.values() %}
{{ value }}{% if not loop.last %},{% endif %}
{% endfor %}

Keys and values from hash:
{% for key,value in hash.items() -%}
{{ key ~ "-" ~ value }}
{% endfor %}
```

When rendered, produces the following:

```
Keys from hash:
 a c b

Values from hash:
1,3,2

Keys and values from hash:
a-1
c-3
b-2
```

Note that Python hashes are not ordered. If order is important, use filters to sort as needed:

File: sorted_control.j2

```
Sorted keys from hash:
{% for key in hash|sort %}
{{ key }}
{% endfor %}

Sorted keys and values from hash:
{% for key,value in hash|dictsort(reverse=True) %}
{{ key ~ "-" ~ value }}
{% endfor %}
```

```
Sorted keys from hash:
a
b
c

Sorted keys and values from hash:
c-3
b-2
a-1
```

For more complex data structures, care must be taken in referencing data, and in passing the right data type to filters within the template; for example:

File: vars

```
---
agents:
  Bryan:
    active: true
    operations:
    - Treadstone
    - Blackbriar
  Sarah:
    active: false
    operations:
    - Buzzsaw
    - Wiremire
    - Brightnote
```

File: conplex_control.j2

```
Keys and values from complex data:
{% for key,value in agents.items() %}
{% if value.active %}+ {% else %}- {% endif %}
Agent({{ loop.index ~ " of " ~ loop.length }}): {{ key }}
{{ value.operations|length }} ⏎
      projects: {{ value.operations|join(', ') }}

{% endfor %}
```

```
Keys and values from complex data:
- Agent(1 of 2): Sarah
3 projects: Buzzsaw, Wiremire, Brightnote

+ Agent(2 of 2): Bryan
2 projects: Treadstone, Blackbriar
```

DEMO: Jinja2 Templates

Data and template rendering at the Python layer
Group creation of template

Python and Jinja2

It is useful to remember the layers of parsing involved when dealing with templates. Ansible gets first pass and will replace things like vault encrypted sections. It then uses the pyYAML parser to verify the playbook is valid YAML. Then it assembles all vars within the calling scope into a hash. When a template needs to be processed, a Jinja template object gets created and the hash is passed to the render method. It is occasionally useful when debugging to go into an interactive Python session and try the same operation. The errors can be better, and the debugging tools are certainly better.

```
$ python
>>> var1=10 # - Integer Object
>>> var2=10.5 # - Float Object
>>> var1 + var2 # - Returns a float value
20.5
>>> int(var1 + var2) # - When converted to Integer,
      the result loses precision
20
>>> var1="boom"
>>> var1*5
'boomboomboomboomboom'
>>> var2="bam"
>>> var1 + var2
'boombam'
>>> from jinja2 import Template
>>> help(Template)
. . . output omitted . . .
>>> t=Template('{{ var1 + var2 }}')
>>> t.render({'var1':10, 'var2':10.5})
u'20.5'
>>> t.render({'var1':'boom', 'var2':'bam'})
u'boombam'
>>> quit()
```

Equivalent template operations from within Ansbile:

```
$ ansible-console localhost
f:5]$ debug msg='msg={{ 10+10.5 }}'
localhost | SUCCESS => {
    "msg": "msg=20.5"
}
[f:5]$ debug msg='{{ 10+10.5 | int }}'
localhost | SUCCESS => {
    "msg": "20"
}
[f:5]$ debug msg="testing{{'123'|replace
      ('1','one')}}"
localhost | SUCCESS => {
    "msg": "testingone23"
}
[f:5]$ exit
```

Group Template Creation Exercise

Instructor drives the keyboard, but entire class suggests login and syntax. Collectively, come up with a template that will give the desired output given the input variables listed.

File: jinja_vars.yaml

```
+ - fruits: ['apple', 'pear', 'orange', 'banana']
+ - old_vers: [1,2,3,4]
+ - new_vers: [3,4,5,6]
+ - hosts:
+     host1:
+       settings:
+         reveal: true
+         appver: 2
+       secret: "turtle"
+     host2:
+       settings:
+         reveal: false
+         appver: 5
+       secret: "fidgit"
+     host3:
+       settings:
+         reveal: true
+         appver: 3
+       secret: "bonnet"
```

File: template_demo.yaml

```
+ ---
+ - hosts: localhost
+   vars_files:
+     - jinja_vars.yaml
+   tasks:
+   - template: dest=/tmp/output src=template.j2
```

Template should produce the following output, and must react to changes to the var data—no fair cheating and just printing the literal strings... :)

File: template_demo.j2

```
+ Help me figure it out!
  . . . snip . . .
```

```
$ ansible-playbook template_demo.yaml
. . . output omitted . . .
$ cat /tmp/output
My sorted list of important fruits: Apple Banana Orange Pear

These versions overlap (in both old and new): [3, 4]

Number of hosts: 3

These are the secrets that can be revealed
      (in no particular order):
bonnet turtle

Checking app version:
host3 is upgraded --> ver:3
host2 is upgraded --> ver:5
host1 is NOT upgraded --> ver:2
```

Lab 5

Estimated Time:
R7: 65 minutes

Task 1: Jinja2 Templates [R7]

Page: 5-20 Time: 20 minutes

Requirements: 💻 (1 station)

Task 2: Jinja2 Templates [R7]

Page: 5-23 Time: 45 minutes

Requirements: 💻 (1 station)

Objectives

- Use templates to enable dynamic expressions and access variables.
- Use template filters to manipulate data
- Use tests to evaluate template expressions
- Use lookups to process data from external sources within templates

Requirements

- (1 station)

Relevance

Lab 5
Task 1
Jinja2 Templates [R7]

Estimated Time: 20 minutes

1) Create a template and corresponding playbook that produces an `/etc/motd` file with contents like those shown here:

```
$ ansible-playbook motd.yaml -b
PLAY [localhost]

TASK [Gathering Facts]
ok: [localhost]

TASK [install figlet]
changed: [localhost]

TASK [register FQDN banner from figlet]
changed: [localhost]

TASK [Generate /etc/motd from template]
changed: [localhost]

PLAY RECAP
localhost                  : ok=4    changed=3    unreachable=0    failed=0
```

File: /etc/motd

```
Hello, and welcome to:


     _         _   _              _  _                                    _
 ___| |_ __ _| |_(_) ___  _ __ | || |   _____  ____ _ _ __ ___  _ __ | | ___   ___ ___  _ __ ___
/ __| __/ _` | __| |/ _ \| '_ \| || |_ / _ \ \/ / _` | '_ ` _ \| '_ \| |/ _ \ / __/ _ \| '_ ` _ \
\__ \ || (_| | |_| | (_) | | | |__   _|  __/>  < (_| | | | | | | |_) | |  __/| (_| (_) | | | | | |
|___/\__\__,_|\__|_|\___/|_| |_|  |_|(_)___/_/\_\__,_|_| |_| |_| .__/|_|\___(_)___\___/|_| |_| |_|
                                                                |_|
Primary IP: 10.100.0.4
On network: 10.100.0.0/24

#######
#
# reboot   system boot  3.10.0-514.2.2.e Mon Aug  7 13:52 - 17:29 (32+03:36)
# reboot   system boot  3.10.0-514.2.2.e Mon Jan 16 12:36 - 17:29 (235+03:52)
# reboot   system boot  3.10.0-514.2.2.e Mon Jan 16 12:34 - 12:36  (00:01)
# reboot   system boot  3.10.0-514.2.2.e Mon Jan 16 11:39 - 12:34  (00:54)
# reboot   system boot  3.10.0-514.2.2.e Fri Jan 13 16:25 - 16:37  (00:12)
#
# wtmp begins Fri Jan 13 16:25:01 2017
#
#######
```

The following requirements must be met:

- FQDN banner, IP, network, etc. must not hardcode info. Detect dynamically.
- Network line can show netmask in "traditional" dotted-quad notation. Bonus points if it uses CIDR notation as show in the sample output.
- Bottom block of output should be a list of the last 10 times the system has rebooted (may be less lines if the system has not rebooted that many times).

Consult the following list of hints if you get stuck:

- `figlet` can be installed from the RPM in `/labfiles`
- playbook makes use of 4 facts
- The Jinja2 "join" filter can take a list and output a string with each list element separated by a specified character.
- The Ansible "ipaddr" filter can perform many transformations on IP information.

- The Ansible "lookup" function can pull data in from external sources (including the output of running a command).
- The Linux "last" command supports a special pseudo-user called `reboot`.
- The Ansible "comment" filter can prefix a line or block with specified characters.

Objectives

- Use templates to enable dynamic expressions and access variables.
- Use template filters to manipulate data
- Use tests to evaluate template expressions

Requirements

- (1 station)

Relevance

Lab 5
Task 2
Jinja2 Templates [R7]

Estimated Time: 45 minutes

Load Balanced Web Cluster

1) Make sure no containers are currently running:

```
$ docker container rm -f $(docker container ps -qa)
. . . output omitted . . .
```

2) Create an Ansible playbook (using the provided stub playbook) that will set up several docker containers to host a load-balanced web cluster, and also create a static inventory file, using an inline Jinja2 template:

```
$ cp /labfiles/provision_lb_web_cluster.yaml.v1 ~/playbooks/provision_lb_web_cluster.yaml
```

File: ~/playbooks/provision_lb_web_cluster.yaml

```
  . . . snip . . .
    - name: Create static inventory
+     copy:
+       dest: "{{ inventory_directory }}/lb_web_inventory"
+       content: |
+         [web]
+         {% for result in web_build.results %}
+         {{ result.stdout }}
+         {% endfor %}
+         [lb]
+         {{ lb_build.stdout }}
```

3) Run the playbook and check to see that new docker containers are running, and that the inventory file exists:

```
$ ansible-playbook provision_lb_web_cluster.yaml
```

```
PLAY [localhost]

TASK [Gathering Facts]
ok: [localhost]

TASK [Start containers for Web]
changed: [localhost] => (item=1)
changed: [localhost] => (item=2)
changed: [localhost] => (item=3)

TASK [Start container for HAProxy]
changed: [localhost]

TASK [create static inventory]
changed: [localhost]

PLAY RECAP
localhost                  : ok=4    changed=3    unreachable=0    failed=0
$ docker container ps
CONTAINER ID     IMAGE                        COMMAND          CREATED              STATUS           NAMES
0226be100a7f     server1:5000/centos-init   "/sbin/init"     55 seconds ago       Up 53 seconds    haproxy_lb
32d082b67cab     server1:5000/centos-init   "/sbin/init"     57 seconds ago       Up 55 seconds    web3
900077c57321     server1:5000/centos-init   "/sbin/init"     59 seconds ago       Up 57 seconds    web2
a88bcd61c81e     server1:5000/centos-init   "/sbin/init"     About a minute ago   Up 58 seconds    web1
$ cat lb_web_inventory
[web]
172.18.0.2  ansible_host=web1 ansible_connection=docker
172.18.0.3  ansible_host=web2 ansible_connection=docker
172.18.0.4  ansible_host=web3 ansible_connection=docker
[lb]
172.18.0.5  ansible_host=haproxy_lb ansible_connection=docker
```

These 4 docker containers will be simulating physical hosts for the rest of the lab. They contain a base Centos image and now need to be configured to do something.

4) Examine the provided playbok that installs and configures the web hosts:

```
$ cp /labfiles/configure_web_cluster.yaml ~/playbooks
$ cat configure_web_cluster.yaml
---
- hosts: web
  gather_facts: No
```

```
tasks:
- name: Enable EPEL repo
  yum:
    name: https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm

- name: Install nginx and needed network package
  yum:
    name: ['nginx', 'iproute']

#Facts needed for template
- setup:

- name: Create index.html
  template:
    src: ./templates/index.html.j2
    dest: /usr/share/nginx/html/index.html

- name: Start nginx
  service: name=nginx state=started enabled=yes
```

5) Create the referenced template:

```
$ mkdir ~/playbooks/templates/
```

File: templates/index.html.j2

```
+ <html>
+ <p>This setup has a load balancer configured</p>
+
+ <ul>The following nodes have been configured to serve this cluster:
+   {% for host in groups.web %}
+   <li>Container Name: {{ hostvars[host].ansible_hostname }}
+   IP: {{ hostvars[host].ansible_default_ipv4.address }} Port: 80</li>
+   {% endfor %}
+ </ul>
+
+ <p>You are currently connected to {{ inventory_hostname }}</p>
+ </html>
```

Note the use of `hostvars[host]` to access the facts for hosts other than the one where the task is being executed.

6) Test the play using the inventory file created by the provisioning playbook:

```
$ ansible-playbook -i lb_web_inventory configure_web_cluster.yaml
PLAY [web]

TASK [Enable EPEL Repo]
changed: [172.18.0.3]
changed: [172.18.0.4]
changed: [172.18.0.2]

TASK [Install nginx and needed network package]
changed: [172.18.0.2] => (item=[u'nginx', u'iproute'])
changed: [172.18.0.4] => (item=[u'nginx', u'iproute'])
changed: [172.18.0.3] => (item=[u'nginx', u'iproute'])

TASK [setup]
ok: [172.18.0.3]
ok: [172.18.0.2]
ok: [172.18.0.4]

TASK [Create index.html]
changed: [172.18.0.2]
changed: [172.18.0.4]
changed: [172.18.0.3]

TASK [Start nginx]
changed: [172.18.0.3]
changed: [172.18.0.4]
changed: [172.18.0.2]

PLAY RECAP
172.18.0.2                 : ok=5    changed=4    unreachable=0    failed=0
172.18.0.3                 : ok=5    changed=4    unreachable=0    failed=0
172.18.0.4                 : ok=5    changed=4    unreachable=0    failed=0
```

7) Connect to one of the nodes and examine the output:

```
$ sudo yum install -y elinks
. . . output omitted . . .
$ elinks -dump 172.18.0.2
   This setup has a load balancer configured
```

```
    The following nodes have been configured to serve this cluster:
    * Container Name: a88bcd61c81e IP: 172.18.0.2 Port: 80
    * Container Name: 900077c57321 IP: 172.18.0.3 Port: 80
    * Container Name: 32d082b67cab IP: 172.18.0.4 Port: 80

  You are currently connected to 172.18.0.2
```

8) Create a new playbook that configures the load-balancer:

File: ~/playbooks/configure_lb.yaml

```
+ ---
+ - hosts: lb
+   tasks:
+   - name: Install HAProxy
+     yum: name=haproxy state=present
+
+   - name: Create HAProxy config
+     template:
+       src: ./templates/haproxy.cfg.j2
+       dest: /etc/haproxy/haproxy.cfg
+
+   - name: Start HAProxy
+     service: name=haproxy state=started enabled=yes
```

9) Create the referenced template:

File: templates/haproxy.cfg.j2

```
+ global
+   daemon
+   maxconn 256
+
+ defaults
+   mode http
+   timeout connect 5000ms
+   timeout client 50000ms
+   timeout server 50000ms
+
+ listen http-in
+   bind *:80
+   {% for host in groups.web %}
+     server {{ hostvars[host]['ansible_hostname'] }} ➥
     {{ hostvars[host]['ansible_default_ipv4']['address'] }}:80
+   {% endfor %}
```

Note the use of the alternate "[' (bracket, quote)" syntax to reference the host vars.

10) Execute the new playbook and examine the results:

```
$ cd ~/playbooks
$ ansible-playbook -i lb_web_inventory configure_lb.yaml
. . . snip . . .
PLAY [lb] *********************************************************************

TASK [Gathering Facts] ********************************************************
ok: [172.18.0.5]

TASK [Install haproxy package] ************************************************
changed: [172.18.0.5]

TASK [Create HAProxy config] **************************************************
changed: [172.18.0.5]

TASK [Start the haproxy service] **********************************************
changed: [172.18.0.5]

PLAY RECAP ********************************************************************
172.18.0.5                 : ok=4    changed=3    unreachable=0    failed=0
```

```
$ docker exec haproxy_lb cat /etc/haproxy/haproxy.cfg
. . . snip . . .
listen http-in
  bind *:80
    server c9789f1e008c 172.18.0.2:80
    server 7185f4528063 172.18.0.3:80
    server 014cba2eccb4 172.18.0.4:80
```

11) Verify the load-balancer is working as expected:

```
$ curl -s 172.18.0.5 | grep current
<p>You are currently connected to 172.18.0.2</p>
$ curl -s 172.18.0.5 | grep current
<p>You are currently connected to 172.18.0.3</p>
$ curl -s 172.18.0.5 | grep current
<p>You are currently connected to 172.18.0.4</p>
```

Nice job. You just provisioned and configured a load-balanced web cluster!

12) Scale up your cluster by manually running the existing playbooks overriding the var that determines the number of web nodes, then configure the new web node, again using the existing playbook:

```
$ ansible-playbook -e number_of_nodes=4 provision_lb_web_cluster.yaml
. . . output omitted . . .
$ ansible-playbook -i lb_web_inventory configure_web_cluster.yaml
. . . output omitted . . .
```

13) Watch the results of repeatedly hitting the load balancer endpoint to see which web nodes are servicing the requests:

```
$ watch elinks 172.18.0.5
. . . snip . . .
You are currently connected to 172.18.0.web_node_ip

# wait for several iterations, taking note of the IP's hit, then press:
Ctrl+C
```

The newly added node is not being used because even though the load balancer config was updated, the HAProxy process running in the container was not

restarted. The correct configuration playbook would notify a handler (covered later) to restart the HAProxy process every time the config changed.

14) Manually restart the docker container and verify the new node is getting traffic:

```
$ docker container restart haproxy_lb
$ watch -d elinks 172.18.0.5

# wait for several iterations, taking note of the IP's hit, then press:
Ctrl+C
```

Bonus: Orchestration, Scaleup

15) Create a playbook that can scale up your cluster more elegantly:

File: playbooks/scaleup.yaml

```
+ ---
+ - hosts: localhost
+   gather_facts: No
+   tasks:
+   - name: Get current number of web nodes
+     shell: >
+       {%raw%} docker ps --filter "name=web." --format "{{.Names}}" | wc -l {%endraw%}
+     register: current_number_of_nodes
+
+ - import_playbook: provision_lb_web_cluster.yaml
+   vars:
+     number_of_nodes: "{{ current_number_of_nodes.stdout|int + scale_num|int|default(1) }}"
+
+ - import_playbook: configure_web_cluster.yaml
+ - import_playbook: configure_lb.yaml

+ - hosts: lb
+   gather_facts: No
+   tasks:
+   - name: Reload config for load balancer
+     systemd:
+       name: haproxy
+       state: restarted
```

- Note the use of `{%raw%}` to prevent the YAML parser from misinterpreting the curly braces intended to be literal data fed to the Go templater used by Docker.
- Note overriding the var which determines the number of nodes when importing the playbook, and also providing a default for the scale number.
- Note that this playbook is actually pretty bad in terms of error handling and all the assets are in dire need of being refactored at this point. Blocks, Roles, Tags, and other properties will let us do this far more robustly, and cleanly later.
- Note that the provisioning is not idempotent and should use the `docker_container` module instead.
- Note that the inventory is not dynamic and related elements should probably be refactored.

16) Restart your "monitoring" command in one terminal and then in another terminal execute your scaleup playbook:

```
[term1]$ watch -d elinks 172.19.0.5
```

```
[term2]$ ansible-playbook -i lb_web_inventory -e scale_num=2 scaleup.yaml
. . . output omitted . . .
```

Cleanup

17) Delete the containers:

```
$ docker container rm -f $(docker ps -qa)
```

Solution to Task 1

18) Hopefully you pushed through it without peeking, but this is one functioning way to accomplish task 1:

File: motd.yaml

```
---
- hosts: localhost
  tasks:
  - name: install figlet
    yum: name=/labfiles/figlet-2.2.5-9.el7.x86_64.rpm
  - name: generate hostname banner with figlet
    command: "figlet -w 150 {{ ansible_fqdn }}"
    register: fig_hostname
  - name: Generate /etc/motd from template
    template:
      src: motd.j2
      dest: /etc/motd
```

File: motd.j2

```
Hello, and welcome to:

{{ fig_hostname.stdout }}
Primary IP: {{ ansible_default_ipv4.address }}
On network: {{ [ansible_default_ipv4.network, ansible_default_ipv4.netmask] | join('/') | ipaddr('net') }}

{{ lookup('pipe', 'last -n10 reboot') | comment('plain', prefix='#######\n#', postfix='#\n#######') }}
```

Content

Chapter 6

TASK CONTROL

Loops

with_items: → over list
with_nested: → nested lists
with_dict: → over hash
loop: **alternative to** with_{items,dict}
- use query function to always return list

Standard Loops

The simplest loop uses the with_items: clause and a provided YAML list to loop over all items in the list. Items from the list are referenced within the task using: {{ item }}. If the list has more complex data-structures like a hash, then the elements are accessed using {{ item.keyname }}. Several modules optimize their operation when called with a list. For example, the package related modules will batch the list together and run a single transaction. This is generally desirable for performance, but can cause problems if the items must be processed as separate transactions for some reason. The following examples show the use of basic loops:

File: standard_loop.yaml

```
- name: Install packages
  package: name={{ item }} state=latest   #NOT best practice anymore
  with_items:
  - vim
  - tmux
```

File: standard_loop2.yaml

```
- name: Create users
  user:
    name: "{{ item.name }}"
    uid: "{{ item.uid }}"
    generate_ssh_key: "{{ item.ssh }}"
    shell: /bin/bash
  with_items:
  - { name: 'bryan', uid: '5000', ssh: 'yes' }
  - { name: 'sarah', uid: '5001', ssh: 'no'  }
```

Nested Loops

A task can iterate over a multidimensional array using the `with_nested:` clause. The following example uses a nested loop to grant each user in a second list ACLs for the directories in the first list:

File: nested_loop.yaml

```
- name: Grant permissions
  acl:
    path: "/data/app1/{{ item[0] }}"
    entity: "{{ item[1] }}"
    etype: user
    permissions: rw
    state: present
  with_nested:
  - ['dev','qa','prod']
  - [ 'bryan', 'dax', 'rob' ]
```

Looping over Hash

Looping over a hash is very similar to the standard loop operating on a hash, with the primary difference being how the values of the hash are referenced. The following example uses a loop over hash construct to output info contained in a hash:

File: hash_loop.yaml

```
---
- hosts: localhost
  vars:
    guns:                diccionario
      224-146372:        key
        make: HK             value.make
        model: VP9           value.model
      C092908:
        make: CZ
        model: Tactical Sport Orange
      DHE306:            key
        make: Glock          value.make
        model: 17C           value.model
  tasks:
  - name:
    debug:
      msg: "serial num: {{ item.key }} is a {{ item.value.make }} {{ item.value.model }}"
    with_dict: "{{ guns }}"
```

Loops (cont.)

with_file: → over contents of file list
with_fileglob: → **over matched file list** glob son los * y ?
with_sequence: → **over number sequence**

Looping over Files

Because it uses a list as the set to operate on (like a standard loop), it is tempting to think that {{ item }} within the loop will be the name of the file. However, the with_file: construct instead returns the contents of each of the files in the list. For example, note how this task installs multiple SSH keys as trusted (the authorized_key module expect the key value, not the name of the file:

File: file_loop.yaml

```
- name: Install authorized keys
  authorized_key:
    user: deploy
    state: present
    key: '{{ item }}'
  with_file:
  - public_keys/id_rsa-app1.pub
  - public_keys/id_rsa-app2.pub
```

Looping over Glob

Consider the previous example, and what it would be like to maintain the list of keys if it grew large, or changed rapidly over time. To loop over a list of file names that match a pattern, use the with_fileglob: construct. If you actually need the contents of each of the files, then you can use a lookup function to read in the content based on the filename as shown in the following example:

File: file_glob_loop.yaml

```
- name: Install authorized keys
  authorized_key:
    user: deploy
    state: present
    key: "{{ lookup('file', '{{ item }}') }}"
  with_fileglob:
    - /public_keys/id_rsa*.pub
```

lookup con file: funcion jinja2 que devuelve el contenido del fichero

Looping over a Sequence

The with_sequence: construct can be used to generate lists and loop over those lists. By default (if used with the count= argument) it produces a simple increasing decimal list of numbers from zero to the number specified. It can produce far more sophisticated lists, including descending, hex/octal, stride intervals, and printf style format strings. The following example uses a list of two sequences to create a number of accounts:

File: sequence_loop.yaml

```
- name: Create test accounts
  user:
    name: "{{ item }}"
    state: present
  with_sequence:
  - "start=1 end=20 format=red%02d"
  - "start=20 stride=2 end=30 format=blue%02d"
```

El formato %02d pone el numero con dos dígitos

Loops and Variables

Over inventory
- {{ groups['group_name'] }}
- {{ play_hosts }}
- with_inventory_hostnames:

until: → **until condition met**

Registered variable from loop structure
- results[...]

Looping over Inventory

The inventory group, or list of hosts, passed via the hosts: construct of a play is essentially an outer loop causing all tasks within the play to run for each host within the list. Sometimes, within a single task, it is useful to loop over some portion of the inventory. This can be done either using the special Ansible groups of play_hosts variables, or the with_inventory_hostname: construct.

The following example constructs a list of all inventory hostnames that are the intersection of the dc1, and web groups, and then uses that list to reference the source directories that are copied to the /data directory on the hosts for this play:

File: inventory_loop.yaml

```
- name: copy
  copy:
    dest: /data
    src: files/{{ item }}
  with_inventory_hostnames:
  - dc1:&web
```

hosts que están a la vez en el grupo dc1 y web

Looping Until Condition

The until: construct can be used to loop until some condition is met. By default the task will wait 5 seconds between each retry, and make 3 attempts. The following example shows looping until the script being executed returns an exit code of zero (waiting up to 5 minutes, and checking each minute):

File: until_loop.yaml

```
- name: Monitor app until initializes
  script: files/app_alive_yet.sh
  register: result
  until: result.rc = 0
  retries: 5
  delay: 60
```

Reading Registered Variables from Loop

When the results of a task are captured with the register: construct within a loop, the data-structure formed is more complicated. A result hash is constructed with several keys. The result key is a list of results from each iteration of the loop. Each list item is a hash with all the normal results from that iteration. For example:

File: registered_loop_structure.yaml

```
- shell: "echo {{ item }}"
   with_items:
   - "one"
   - "two"
   register: result
```

```
"result": {
        "changed": true,
        "msg": "All items completed",
        "results": [
            {
                ...
                "changed": true,
                "cmd": "echo one",
                "delta": "0:00:00.003689",
                "end": "2017-08-24 16:43:44.624793",
                "failed": false,
                ...
                "item": "one",
                "rc": 0,
                "start": "2017-08-24 16:43:44.621104",
                "stderr": "",
                "stderr_lines": [],
                "stdout": "one",
                "stdout_lines": [
                    "one"
                ]
            },
            {
                ...
                "changed": true,
                "cmd": "echo two",
                "delta": "0:00:00.003547",
                "end": "2017-08-24 16:43:44.815693",
                "failed": false,
                ...
                "item": "two",
                "rc": 0,
                "start": "2017-08-24 16:43:44.812146",
                "stderr": "",
                "stderr_lines": [],
                "stdout": "two",
                "stdout_lines": [
                    "two"
                ]
            }
        ]
```

DEMO: Constructing Flow Control

Use `with_nested:`
Use `with_items:`
Use `with_inventory_hostnames:`

Prep

Add a few users to demo with (pick whatever usernames you want – sample output uses the names specified here):

```
[guru@station1]$ for i in bryan dax cody; do sudo useradd $i; done
```

Nested Looping

As discussed, a task can iterate over a multidimensional array using the `with_nested:` clause. Let's use a standard loop to create some directories based on a list stored in a variable, then grant a second list users ACLs for each of the directories:

File: `nested_loop.yaml` cp /labfiles/solutions/chap6/nested_loop.yaml.v* .

```
+ ---
+ - hosts: localhost
+   vars:
+     environments: ['dev','qa','prod']
+   tasks:
+   - name: Create dirs
+     file: path=/data/app1/{{ item }} state=directory
+     with_items:
+     - "{{ environments }}"
```

probar la v2 que tiene otro ejemplo de buble con dos listas

Test the playbook and verify it created the expected directories:

```
$ ansible-playbook nested_loop.yaml -b
. . . output omitted . . .
$ find /data
data
/data/app1
```

```
/data/app1/dev
/data/app1/qa
/data/app1/prod
```

Now we add a task that uses a nested loop for granting the permissions:

File: nested_loop.yaml /labfiles/solutions/chap6/nested_loop.yaml.v2

```
+   - name: Grant permissions
+     acl:
+       path: "/data/app1/{{ item[0] }}"
+       entity: "{{ item[1] }}"
+       etype: user
+       permissions: rw
+       state: present
+     with_nested:
+     - "{{ environments }}"
+     - [ 'bryan', 'dax', 'cody' ]
```

Run the playbook and verify the results:

```
$ ansible-playbook nested_loop.yaml -b
. . . snip . . .
TASK [Grant permissions]
changed: [localhost] => (item=[u'dev', u'bryan'])
changed: [appserver] => (item=[u'dev', u'dax'])
changed: [appserver] => (item=[u'dev', u'cody'])
changed: [appserver] => (item=[u'qa', u'bryan'])
changed: [appserver] => (item=[u'qa', u'dax'])
changed: [appserver] => (item=[u'qa', u'cody'])
changed: [appserver] => (item=[u'prod', u'bryan'])
changed: [appserver] => (item=[u'prod', u'dax'])
changed: [appserver] => (item=[u'prod', u'cody'])

PLAY RECAP
localhost                  : ok=3    changed=2    unreachable=0    failed=0
$ getfacl /data/app1/dev/
user::rwx
user:bryan:rw-
user:dax:rw-
user:cody:rw-
group::r-x
mask::rwx
other::r-x
```

Looping over Inventory ESTA PARTE DE LA DEMO HABRIA QUE HACERLA EN SERVER1 EN LA CLASE NO LA HAREMOS

As root on the classroom server. Demonstrate looping over inventory groups, by first creating some new groups in the classroom server's inventory file (add all hosts in use to one of the two groups):

File: /etc/ansible/hosts EDITAR EL FICHERO /etc/ansible/hosts DE SERVER1. HACERLO CON STATION1 Y STATION2

```
   server1.example.com ansible_connection=local
   [stations]
   station[1:X].example.com

+  [odd]
+  station1.example.com
+  station3.example.com
   . . . snip . . .

+  [even]
+  station2.example.com
+  station4.example.com
   . . . snip . . .
```

Now we create a playbook that runs a task on each even host causing it to ping every odd host, and register the result. Note the use of the ignore_errors: true clause which prevents the play from exiting if a particular iteration of the loop fails because the host is down:

File: inventory_loop.yaml ESTÁ EN /root/GL380-couse-files/demo/solutions/inventory_loop.yaml.v1

```
+  ---
+  - hosts: even
+    tasks:
+    - name:
+      command: "ping -c1 {{ item }}"
+      with_inventory_hostnames:
+      - odd
+      ignore_errors: true
+      register: pings

+    - debug: var=pings
```

Execute the playbook and examine the results (especially the structure of the results variable):

```
[root@server1]# ansible-playbook inventory_loop.yaml
. . . output omitted . . .
```

Create a playbook that can randomly add firewall rules on stations in the odd inventory group (the one's that will be pinged), that blocks ICMP from a single even host's IP. Start by testing the logic using just the debug module:

File: random_icmp_block.yaml

```
+ - hosts: odd
+   gather_facts: No
+   tasks:
+   - debug:
+       msg: "{{ inventory_hostname }} blocks {{ groups['even']|random }}"
+     when: 100|random >= 50
+     with_sequence: count=5
```

CON SOLO DOS ESTACIONES EL RANDOM NO TIENE MUCHO SENTIDO
NO MERECE LA PENA HACERLO PERO REVISAR EL CODIGO

Test the playbook to verify the referenced variable, conditional clause, and loop produce the desired results:

```
# ansible-playbook random_icmp_block.yaml
. . . snip . . .
ok: [station5.example.com] => (item=2) => {
    "msg": "station5.example.com will block station2.example.com"
}
skipping: [station5.example.com] => (item=3)
. . . snip . . .
```

Modify the playbook so it creates actual firewall rules, then execute:

File: random_icmp_block.yaml

```
    tasks:
-   - debug:
-       msg: "{{ inventory_hostname }} will block {{ groups['even']|random }}"
+   - iptables:
+       chain: INPUT
+       source: "{{ groups['even']|random }}"
+       protocol: icmp
+       jump: REJECT
      when: 100|random >= 50
→     with_sequence: count=52
```

```
# ansible-playbook random_icmp_block.yaml
. . . output omitted . . .
# ansible odd -a "iptables -L INPUT"  #examine results
station3.example.com | CHANGED | rc=0 >>
Chain INPUT (policy ACCEPT)
```

```
target     prot opt source               destination
REJECT     icmp --  station2.example.com  anywhere      reject-with icmp-port-unreachable
. . . snip . . .
```

This arrangement simulates what might happen within a complex environment where individual firewall rules have been deployed incorrectly or certain hosts are simply down. Imagine wanting to see if all webservers can connect to all database servers.

Add the following task to the playbook to collect the results into a local file:

File: inventory_loop.yaml ESTÁ EN /root/GL380-couse-files/demo/solutions/inventory_loop.yaml.v2

```
+   - lineinfile:
+       path: /tmp/failed_pings
+       create: Yes
+       line: "{{ ansible_fqdn }} --> {{ item.item }}"
+       create: yes
+     when: item.rc != 0
+     with_items: "{{ pings.results }}"
+     delegate_to: localhost
```


Note the use of the `when:` clause which only runs the task for result items that failed, and the `delegate_to:` clause which causes the module to execute locally instead of on the even hosts. The `ansible_fqdn` variable is from facts and refers to the even host we are processing. The `item.item` refers to the item key of the result item generated by the `with_items:` construct.

Execute the playbook and examine the results:

```
# ansible-playbook inventory_loop.yaml
. . . output omitted . . .
# cat /tmp/failed_pings
station4 --> station3.example.com
station2 --> station5.example.com
. . . snip . . .
```

Cleanup

Be sure the firewall rules created earlier are removed. Process all stations (not just odd) just to be sure:

```
[root@server1]# ansible stations -m iptables -a "chain=INPUT flush=true"
```

Conditionals

`when:` → **only perform task if condition is true**
`when:` **processed for each item in loop**
Can be used with
- `import*`, `include*`
- roles

`fail`**, and** `assert` **exit play based on conditions**

Útil para implementar la IDEMPOTENCIA en módulos que carecen de ella

Skipping Tasks

Most Ansible modules are idempotent and can safely be called as many times as you want. If the module detects that the system is already in the correct state, it generally skips doing any further work, and reports no change. Some modules (especially command modules) have no good way to detect if they should run. The `creates:` clause solves the problem if the presence of a file can determine if the command has already executed. For more complicated cases, a `when:` clause can evaluate expressions and then the task can run only when the conditions are true.

When clauses generally compare data stored in variables (often registered from an earlier task), or alternatively from facts registered by the setup module. The comparisons support (un)equality, logical and other operators; see http://jinja.pocoo.org/docs/dev/templates/#comparisons for details. The following examples give a sense of what is possible:

File: conditionals.yaml

```
- name: Shutdown Centos 6
  command: /sbin/shutdown -t now
  when:
  - ansible_distribution == "Centos"
  - ansible_distribution_major_version == "6"

- name: Shutdown Centos 7
  command: systemctl poweroff
  when:
  - ansible_distribution == "Centos"
  - ansible_distribution_major_version == "7"
```

File: conditionals2.yaml

```
- fail:
    msg: "System doesn't meet needed specs"
  when:
    - ansible_memtotal_mb < 15999
    - ansible_processor_cores < 4
```

File: conditionals3.yaml

```
- name: Test if user can sudo
  command: /usr/bin/sudo -v
  register: sudo_test
  ignore_errors: true
- name: Fail if user can't sudo
  fail: msg="User can't sudo :("
  when: sudo_test.rc == 1
```

ignore_errors PERMITE EN CUALQUIER TASK EVITAR QUE TERMINE CON ERROR
EN ESTE CASO EL ÚNICO FIN DE LA TASK ES VER QUE EL USUARIO PUEDE HACER sudo INDEPENDIENTEMENTE DE QUE FUNCIONE O NO

File: conditionals4.yaml

```
- acl: path=/tmp/file1
    register: file1_acl
- assert:
    that:
    - '"group:admin:rw-" in file1_acl.acl'
    - '"user:sarah:r--" in file1_acl.acl'
```

assert TERMINA LA TASK CON EXITO SI EXISTEN LAS ACL INDICADAS EN LA VARIABLE DE LA TASK ANTERIOR

DEMO: Conditionals

Grow a volume and filesystem only if there is room.

Grow - All Values Hardcoded

It is generally best to start with a simple playbook and then once basic functionality has been verified, it can be modified with additional checks, etc. First we create a small LV that we can play with:

HACERLO EN STATION1

```
[guru@station1]$ ansible localhost -m lvol↵
      -a "lv=demo2 vg=vg0 size=100M" -b
$ sudo lvs vg0/demo2
  LV    VG  Attr       LSize
  demo2 vg0 -wi-a----- 100.00m
```

Now a bare-bones playbook that can grow the volume:

File: ~/playbooks/grow_lv.yaml

/labfiles/solutions/chap6/grow_lv.yaml.v1

```
+ ---
+ - hosts: localhost
+   become: Yes
+   tasks:
+   - name: Grow LV
+     lvol: "vg=vg0 lv=demo2 size=105M"
```

```
$ ansible-playbook grow_lv.yaml
. . . snip . . .
localhost     : ok=2    changed=1    unreachable=0    failed=0
$ sudo lvs vg0/demo2
  LV    VG  Attr       LSize
  demo2 vg0 -wi-a----- 108.00m
```

Modify the playbook so that the LV and size are requested by prompt:

File: ~/playbooks/grow_lv.yaml

/labfiles/solutions/chap6/grow_lv.yaml.v2

```
    ---
    - hosts: localhost
      become: Yes
+     vars_prompt:
+     - name: "LV"
+       prompt: "What LV do you want to grow?"
+       private: no
+     - name: "SIZE"
+       prompt: "How big should LV be in MB?"
+       private: no
      tasks:
      - name: Grow LV
→       lvol: "vg=vg0 lv={{ LV }} size={{ SIZE }}"
```

Re-run the playbook and verify it resizes the LV correctly:

```
$ ansible-playbook grow_lv.yaml
What LV do you want to grow?: demo2
How big should LV be in MB?: 110
. . . output omitted . . .
$ sudo lvs vg0/demo2
  LV    VG  Attr       LSize
  demo2 vg0 -wi-a----- 112.00m
```

What if the user tries to grow a volume that doesn't exist?

```
$ ansible-playbook grow_lv.yaml
What LV do you want to grow?: foo
How big should LV be in MB?: 100
. . . snip . . .
localhost     : ok=2    changed=1    unreachable=0    failed=0
$ sudo lvs | grep foo
  foo         vg0 -wi-a----- 100.00m
```

Not good! It created a new volume instead. Let's modify the playbook to fail if the volume doesn't already exist:

File: ~/playbooks/grow_lv.yaml

```
    tasks:      /labfiles/solutions/chap6/grow_lv.yaml.v3
+   - name: Verify LV exists
+     assert:
+       that: LV in ansible_lvm.lvs
+       msg: "Can't grow LV that doesn't exist"
    - name: Grow LV
      lvol: "vg=vg0 lv={{ LV }} size={{ SIZE }}"
```

Test again with a bogus LV name:

```
$ sudo lvremove vg0/foo
Do you really want to remove logical volume vg0/foo? [y/n]: y
  Logical volume "foo" successfully removed
. . . output omitted . . .
$ ansible-playbook grow_lv.yaml
What LV do you want to grow?: foo
How big should LV be in MB?: 100
TASK [Verify LV exists]
fatal: [localhost]: FAILED! => {
    "assertion": "LV in ansible_lvm.lvs",
    "changed": false,
    "evaluated_to": false,
    "failed": true,
    "msg": "Can't grow LV that doesn't exist"
}
```

What if the VG doesn't have enough free space for the requested size? Let's modify the playbook again to request the growth size, and to fail with a nice message if the VG does not have enough free space:

File: ~/playbooks/grow_lv.yaml

```
+     prompt: "How big should LV be in MB?"
→   - name: ~~"SIZE"~~"GROW"
→     prompt: "~~How big should LV be in~~Grow by how many MB?"
      private: no      /labfiles/solutions/chap6/grow_lv.yaml.v4
    tasks:             ESTÁ MODIFICADO PARA QUE FUNCIONE
    - name: Verify LV exists
      assert:
        that: LV in ansible_lvm.lvs
        msg: "Can't grow LV that doesn't exist"
+   - name: Get free space in VG
+     command: >
+           vgs -o vg_free --noheadings
+           --nosuffix --units m vg0
+     register: vgs_result
+   - fail:
+       msg: "Not enough free space in VG
+        to grow by {{ GROW }} MB"
+     when: vgs_result.stdout|trim|int < GROW|int
```

Hmmm. Now that we are prompting for the growth amount, we need to compute the final size to be that plus the current size. Unfortunately the facts variable stores the size in GB and rounds to a precision where we could miscalculate. Better to just query via **lvs** and use that output. First we examine the output so we know what munging it might need:

```
$ sudo lvs -o lv_size --noheadings --nosuffix
      --units m vg0/demo2
    112.00
```

Now we adjust the playbook to collect the current size and then compute the final size:

File: ~/playbooks/grow_lv.yaml /labfiles/solutions/chap6/grow_lv.yaml.v5

```
+     - name: Get current size of LV
+       command: >
+            lvs -o lv_size --noheadings
+            --nosuffix --units m vg0/{{ LV }}
+       register: lvs_result
      - name: Grow LV
→       lvol: "vg=vg0 lv={{ LV }} size={{ ~~SIZE~~ lvs_result.stdout.split('.')[0]|trim|int + GROW|int }}"
```

Time to test the playbook again:

```
$ ansible-playbook grow_lv.yaml
What LV do you want to grow?: demo2
Grow by how many MB?: 5
localhost        : ok=5    changed=3    unreachable=0    failed=0

$ sudo lvs vg0/demo2
  LV    VG  Attr       LSize
  demo2 vg0 -wi-a----- 120.00m
```

What if the VG name is different than the one currently in the playbook. Perhaps we could look it up in the collected facts, based on the LV name:

File: ~/playbooks/grow_lv.yaml /labfiles/solutions/chap6/grow_lv.yaml.v6

```
      - name: Get current size of LV
→       command: lvs -o lv_size --noheadings --nosuffix --units m ~~"vg0/"~~{{ ansible_lvm['lvs'][LV]['vg'] }} + "/" + LV }}
        register: lvs_result
      - name: Grow LV
→       lvol: "vg=~~vg0~~{{ansible_lvm['lvs'][LV]['vg']}} lv={{LV}} size={{lvs_result.stdout.split('.')[0]|trim|int + GROW|int}}"
```

Handlers

Modules report when a change is made
Tasks can trigger actions on change with `notify:`
Handlers run one at end of play

Handlers

Most modules are idempotent and will only make a change to the system when needed. Each task then reports if a change was made. For command modules the `changed_when:` clause can used to determine when a change is reported. Any task can have an optional `notify:` clause that will signal Ansible to run one or more handlers (just tasks). These handlers are run only once at the end of the play, regardless of how many times a notification was generated during the play.

The following examples show syntactically how handlers and notifications are used within a playbook:

File: handler1.yaml

```
tasks:
  - name: Install web config
    template: src=files/httpd.j2 dest=/etc/httpd/httpd.conf
    notify:
      - restart Apache

handlers:
  - name: restart Apache
    service: name=httpd state=restarted
```

Starting with Ansible 2.2, handlers can be tied to "listen" on a named topic allowing them to be triggered by a name other than the one specified in the `name:` clause. For example:

File: handler2.yaml

```
tasks:
  - name: Configure web server
    copy: dest=/etc/httpd/conf.d/ src=files/site.conf
    notify: restart webstack
  - name: Configure MySQL
    copy: dest=/etc/mysql/my.cnf src=files/my.conf
    notify: restart webstack

handlers:
  - name: restart Apache
    service: name=httd state=restarted
    listen: restart webstack
  - name: restart mysql
    service: name=mariadb state=restarted
    listen: restart webstack
```

CON LISTEN EL HANDLER ESTA ATENTO AL EVENTO DIRECTAMENTE

UTIL PARA REALIZAR UNA ACCIÓN NECESARIA DESPUES DE OTRA

Tags

Label(s) assigned to a task

- allows tasks to be skipped or run

ansible-playbook foo.yaml --tags "tag1,tag2,..."
ansible-playbook foo.yaml --skip-tags "tag1,..."
Special tags

- always
- tagged
- untagged

Tags

It is generally best to organize playbooks so that they accomplish a logical task. This helps them to be reusable, and easy to read and understand. When a larger set of play needs to be applied, a "container" playbook can be created that uses include clauses to pull in the needed individual plays.

Sometimes however, it makes sense to package tasks within a play but also allow for a subset of those task to be run. For example perhaps the same playbook can be used to install an application's foundations, the app itself, and maybe also include orchestration tasks for things like restarting or upgrading the app. In this case, tags might be applied to the tasks in such a way that each of these different behaviors can be selected. Consider the following example:

```
$ ansible-playbook tags.yaml
      #runs all tasks as if "--tags all" had been used
$ ansible-playbook tags.yaml --tags base
      #runs the first and last tasks
$ ansible-playbook tags.yaml --skip-tags base
      #runs all but first task
$ ansible-playbook tags.yaml --tags untagged
      #runs only the second to last task
$ ansible-playbook tags.yaml --tags restart
      #runs the service task and both debug tasks
```

File: tags.yaml

```
- yum: name={{ item }} state=installed
  with_items: [ 'frob', 'grok', 'hork' ]
  tags:
  - base
- copy: src=files/configs dest=/etc
  tags:
  - config
- template: src=templates/hork.j2
    dest=/etc/horkinator.conf
  tags:
  - config
- service: name={{ item }} state=restarted
  with_items: [ 'frob', 'grok', 'hork' ]
  register: result
  tags:
  - restart
- debug: var=result
  tags:
  - restart
- wait_for: port=55355
- debug: msg="Don't hork the frobulator, do you grok that?"
  tags:
  - always
```

Handling Errors

Blocks

- easily apply `when:` to a set of tasks
- handle errors

`block:`, `rescue:`, `always:`

- tasks in block run
- if error encountered then run rescue tasks
- tasks in always run regardless of if block or rescue had errors

Blocks

There are a few cases where it is useful to group tasks together within a play. The first use of blocks is as a convenience to allow a `when:` clause to be listed only once and still applied to the set of enclosed tasks. In this case, the clause is still evaluated independently for each task, but need only be listed once. For example (assume the boolean var `frobstate` exists):

File: `block.yaml`

```
tasks:
- name: Activate Frobulator
  block:
    - yum: name=frob state=latest
    - template:
        src: templates/frob.j2
        dest: /etc/frobulator/frob.conf
    - service: name=frob state=started
  when: frobstate
```

BLOCK PERMITE AGRUPAR UNA LISTA DE TAREAS BAJO UN CONDICIONAL

Error Handling in Blocks

The more significant use case for blocks it to allow for a block of tasks to be run and then trigger a different set of tasks, enclosed in a rescue, if any error is encountered in the first set. The set of tasks in the rescue might send notifications of the failure, or might even back out all the changes made in the block, restoring the system to a sane state. This construct also allows a third block to be specified that always runs. The following example taken from the documentation shows this construct:

File: `block2.yaml`

```
tasks:
- name: Attempt and graceful roll back
  block:
    - debug: msg='I execute normally'
    - command: /bin/false
    - debug: msg='I never execute,
       due to the above task failing'
  rescue:
    - debug: msg='I caught an error'
    - command: /bin/false
    - debug: msg='I also never execute :-('
  always:
    - debug: msg="this always executes"
```

TAMBIEN SE PUEDE HACER BLOCK PARA AÑADIR UNA SECCION RESCUE QUE SE USA CUANDO SE PROVOCA UN ERROR Y TAMBIEN PUEDE LLEVAR UNA SECCIÓN ALWAYS

Lab 6

Estimated Time:
R7: 60 minutes

Task 1: Task Control [R7]

Page: 6-21 Time: 60 minutes

Requirements: 💻 (1 station)

Objectives

- Use handlers to make playbook execution more efficient
- Use when clauses to evaluate registered information and conditionally execute additional tasks
- Use blocks to logically group tasks and improve error handling

Requirements

(1 station)

Relevance

Lab 6
Task 1
Task Control [R7]

Estimated Time: 60 minutes

EL LABORATORIO ACTUALMENTE NO FUNCIONA POR CULPA DE LA VERSIÓN DE PIP.
SE PUEDE COMENTAR EL CÓDIGO DE LA VERSIÓN 10

ESTE CODIGO TRABAJA SOBRE DOS CONTENEDORES UNO CON REDIS (BASE DE DATOS EN RAM) Y OTRO CON UNA APP QUE LO USA

1) Create a directory structure to hold the files for this exercise:

```
[guru]$ cd ~/playbooks
$ mkdir -p control/{tasks,files,handlers,vars}
$ mkdir -p control/files/{redis,app}
$ cd control
```

2) Create a simple playbook that installs the redis package onto a host named redis:

File: playbooks/control/main.yaml /labfiles/solutions/chap6/main.yaml.v1

```
+ ---
+ - hosts: redis
+   tasks:
+   - name: install redis
+     yum: name=redis state=latest
```

3) Create two containers to act as managed hosts, configure Ansible to use the config that allows management of containers, and verify it can connect to both:

```
$ sh /labfiles/docker_centos_init.sh redis
$ sh /labfiles/docker_centos_init.sh app
$ export ANSIBLE_CONFIG=~/docker/ansible.cfg
$ ansible redis,app -m ping
app | SUCCESS => {
    "changed": false,
    "failed": false,
    "ping": "pong"
}
```

export ANSIBLE_CONFIG=/home/guru/docker/ansible.cfg

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

4) Try running the playbook against the redis host:

```
$ ansible-playbook main.yaml
. . . snip . . .
TASK [Install redis]
fatal: [redis]: FAILED! => {. . . snip . . .["No package matching 'redis' found available, installed or updated"]}
```

The repos available to the system do not contain redis, so the installation failed.

5) Instead of creating the needed repo on the managed host, assume it is disconnected from the Internet and that we must provide the files manually and install. Start by getting the `redis` package (and dependencies) on our control host:

```
$ cp /labfiles/{redis,jemalloc}*.rpm files/redis/
$ ls files/redis/
jemalloc-3.6.0-1.el7.x86_64.rpm  redis-3.2.12-2.el7.x86_64.rpm
```

COPIAR ESTOS FICHEROS PARA QUE PUEDAN SER INSTALADOS

6) Modify the play to instead copy the RPMs to the host and then install. Then test the playbook:

File: `main.yaml` /labfiles/solutions/chap6/main.yaml.v2

```
   ---
   - hosts: redis
     tasks:
-    - name: install redis
-      yum: name=redis state=latest
+    - name: copy rpms to host
+      copy: src=files/redis/ dest=/tmp
+    - name: install redis
+      yum: "name=/tmp/{{ item | basename }}"
+      with_fileglob: files/redis/*rpm
```

```
$ ansible-playbook main.yaml
```

```
. . . snip . . .
TASK [Install Redis]
[DEPRECATION WARNING] ... can be safely ignored (see note below)
changed: [redis] => (item=[u'/tmp/jemalloc-3.6.0-1.el7.x86_64.rpm', u'/tmp/redis-3.2.3-1.el7.x86_64.rpm'])
. . . snip . . .
```

Note that the automated suggestion of using the query function to provide a list directly to the yum module will not work in this case.

7) Extend the play so that it adjusts the redis config to listen on all interfaces and starts the service, then execute the play:

File: main.yaml /labfiles/solutions/chap6/main.yaml.v3

```
      with_fileglob: files/redis/*rpm
+   - name: listen on all interfaces
+     lineinfile:
+       regexp: '^bind 127.0.0.1'
+       state: absent
+       dest: /etc/redis.conf
+   - name: start redis
+     service: name=redis state=started enabled=yes
```

PASAR DIRECTAMENTE AL PASO 11 CON LA V5 QUE TIENE LO NECESARIO PARA OPERAR CON LA BASE DE DATOS

```
$ ansible-playbook main.yaml
. . . output omitted . . .
```

8) Install the redis package on the control host to further test that the server is healthy:

```
$ sudo yum install -y files/redis/*rpm
. . . snip . . .
Installed:
  redis.x86_64 3.2.12-2.el7

Dependency Installed:
  jemalloc.x86_64 0:3.6.0-1.el7

Complete!
```

IMPORTANTE:
instalar redis directamente en station1 para poder testear que el servicio funciona en el contenedor

9) Test the Redis server using the Redis CLI tools:

```
$ redis-cli -h 172.18.0.2
172.18.0.2:6379> ping
(error) DENIED Redis is running in protected mode . . . snip . . .
172.18.0.2:6379> quit
```

10) Modify the play to disable protected mode, and re-run:

File: main.yaml /labfiles/solutions/chap6/main.yaml.v4

```
      regexp: '^bind 127.0.0.1'
      state: absent
      dest: /etc/redis.conf
+   - name: disable protected mode
+     lineinfile:
+       regexp: '(^protected-mode) yes'
+       backrefs: yes
+       line: '\1 no'
+       state: present
+       dest: /etc/redis.conf
    - name: start redis
      service: name=redis state=started enabled=yes
```

ESTE CÓDIGO ESTÁ MODIFICADO
EN LOS LABS PARA QUE FUNCIONE

```
$ ansible-playbook main.yaml
. . . snip . . .
TASK [disable protected mode]
changed: [redis]

TASK [start redis]
ok: [redis]
. . . snip . . .
```

Note that even though the config was changed, the task to start the service did not report changed because the service was already started.

11) Re-factor the playbook so that the tasks that modify the config call an included handler instead:

File: main.yaml /labfiles/solutions/chap6/main.yaml.v5

```
   ---
   - hosts: redis
     tasks:
     - name: copy rpms to host
       copy: src=files/redis/ dest=/tmp
     - name: install redis
       yum: "name=/tmp/{{ item | basename }}"
       with_fileglob: files/redis/*rpm
     - name: configure to listen on all interfaces
       lineinfile:
         regexp: "^bind 127.0.0.1$"
         state: absent
         dest: /etc/redis.conf
+      notify: restart redis
     - name: disable protected mode
       lineinfile:
         regexp: "(^protected-mode) yes"
         backrefs: yes
         line: "\1 no"
         state: present
         dest: /etc/redis.conf
+      notify: restart redis
→    - name: startenable redis
→    service: name=redis state=started enabled=yes

+    handlers:
+    - include: handlers/main.yaml
```

12) Create the handler file with a simple task to restart the redis service:

File: handlers/main.yaml

```
+   ---
+   - name: restart redis
+     service: name=redis state=restarted
```

ESTE FICHERO ES PEQUEÑO Y NO ESTÁ EN LA CARPETA DE LABFILES
CREARLO MANUALMENTE

13) Running the playbook on an unconfigured system would now work, but since this is already configured, the handler still won't trigger (even though the service needs to be restarted). Use an ad hoc command to force a restart instead:

EL SERVICIO SI LANZAMOS EL PLAYBOOK DESDE CERO SE REINICIALIZARIA PERO COMO YA ESTA CONFIGURADO NI SIQUIERA LO LANZA EL HANDLER POR LO QUE LO REINICIAMOS CON UN COMANDO AD-HOC

```
$ ansible redis -m service -a "name=redis state=restarted"
. . . output omitted . . .
```

14) Test that the service is working by connecting from the control host using the redis CLI:

TESTEA QUE EL SERVICIO REDIS FUNCIONA

```
$ redis-cli -h 172.18.0.2
PONG
172.18.0.2:6379> set test value1
OK
172.18.0.2:6379> get test
"value1"
172.18.0.2:6379> del test
(integer) 1
172.18.0.2:6379> incr hits
(integer) 1
172.18.0.2:6379> incr hits
(integer) 2
172.18.0.2:6379> del hits
(integer) 1
172.18.0.2:6379> quit
```

15) The playbook is robust, and idempotent, but could be optimized. Change the playbook to meet the following conditions:

/labfiles/solutions/chap6/main.yaml.v6

- All current tasks are contained within a block
- A single task runs before the block that executes the 'redis-cli ping' command to test if redis is already configured correctly
- The "ping test" command must never report changed
- The block should only run if the return code of the ping test is non-zero

16) The result of running the modified playbook should look like the following:

```
$ ansible-playbook main.yaml
```

```
PLAY [redis]

TASK [Gathering Facts]
ok: [redis]

TASK [redis already good?]
ok: [redis]

TASK [copy rpms to host]
skipping: [redis]

TASK [install redis]
skipping: [redis] => (item=[])

TASK [configure to listen on all interfaces]
skipping: [redis]

TASK [disable protected mode]
skipping: [redis]

TASK [enable redis]
skipping: [redis]

PLAY RECAP
redis                      : ok=2    changed=0    unreachable=0    failed=0
```

Installing App

17) Create a directory along with a simple Python requirements file:

```
$ mkdir ~/playbooks/control/app
```

ESTE FICHERO ES PEQUEÑO Y NO ESTÁ EN LA CARPETA DE LABFILES
CREARLO MANUALMENTE

File: app/requirements.txt

```
+ flask
+ redis
```

18) Create a simple Flask app that reports a running total of the number times it is accessed (stored in the redis instance).

File: app/app-v1.py /labfiles/solutions/chap6/app-v1.py

```
+ from flask import Flask
+ from redis import Redis
+ import os
+ app = Flask(__name__)
+ redis = Redis(host='redis', port=6379)

+ @app.route('/')
+ def hello():
+   redis.incr('hits')
+   return 'Hello from host %s. Site accessed %s times.\n' % (os.environ.get('HOSTNAME'),redis.get('hits'))

+ if __name__ == "__main__":
+   app.run(host="0.0.0.0", debug=True)
```

19) Create a simple vars file that records the version of the app we want to deploy:

File: vars/app-version

```
+ ---
+ deploy_version: 1
```

ESTE FICHERO ES PEQUEÑO Y NO ESTÁ EN LA CARPETA DE LABFILES
CREARLO MANUALMENTE

20) Add a new play to the existing playbook that deploys the app onto another managed host (use the provided file from /labfiles): /labfiles/solutions/chap6/main.yaml.v7

```
$ cp /labfiles/CHAP6_main.yaml.v7 /home/guru/playbooks/control/main.yaml
$ less main.yaml
. . . output omitted . . .
```

21) Populate the app/files with the needed RPMs and test execution of the play:

```
$ yumdownloader --resolve --destdir=files/app python python-pip
. . . snip . . .
(1/2): python-2.7.5-48.el7.x86_64.rpm                                    |  90 kB  00:00:00
(2/2): python-pip-7.1.0-1.el7.noarch.rpm                                 | 1.5 MB  00:00:0
$ ansible-playbook main.yaml --limit app
PLAY [redis]
skipping: no hosts matched
```

```
PLAY [app]

TASK [Gathering Facts]
ok: [app]

TASK [copy rpms to host]
changed: [app]

TASK [install Python]
changed: [app] => (item=[u'/tmp/python-2.7.5-48.el7.x86_64.rpm', u'/tmp/python-pip-7.1.0-1.el7.noarch.rpm'])

TASK [install requirements]
changed: [app] => (item=flask)
changed: [app] => (item=redis)

TASK [install app version 1]
changed: [app]
```

22) This service needs a systemd unit file so that it can be managed. Create a template:

File: app/app.service.j2 /labfiles/solutions/chap6/app.service.jd

```
+ [Unit]
+ Description=app1 daemon
+ After=network.target

+ [Service]
+ Environment=HOSTNAME={{ ansible_fqdn }}
+ ExecStart=/usr/bin/python /root/app.py

+ [Install]
+ WantedBy=multi-user.target
```

LAS PLANTILLAS JINJA2 USAN VARIABLES PARA GENERALIZAR EL CONTENIDO DEL FICHERO. EN ESTE CASO {{ansible_fqdn}} SE SUBSTITUYE POR EL NOMBRE DEL HOST

23) Extend the play (v8) adding tasks to install the unit file from template, create a link to the deployed version, and start the app:

File: main.yaml /labfiles/solutions/chap6/main.yaml.v8

```
    - name: "install app version {{ deploy_version }}"
      copy:
        src: app/app-v{{ deploy_version }}.py
        dest: /root/
+   - name: install unit file
+     template:
+       src: app/app.service.j2
+       dest: /etc/systemd/system/app.service
+   - name: link to app version {{ deploy_version }}
+     file:
+       state: link
+       dest: /root/app.py
+       src: /root/app-v{{ deploy_version }}.py
+   - name: start app
+     systemd:
+       name: app
+       daemon_reload: Yes
+       state: started
+       enabled: Yes
```

24) Execute the play and verify that the app functions by connecting a few times with curl:

```
$ ansible-playbook main.yaml --limit app
. . . snip . . .
TASK [install unit file]
changed: [app]

TASK [link to app version 1]
changed: [app]

TASK [start app]
changed: [app]

PLAY RECAP
app                        : ok=8    changed=3    unreachable=0    failed=0
$ curl 172.18.0.3:5000
Hello from host 1955223036ff. Site accessed 1 times.
$ curl 172.18.0.3:5000
```

```
Hello from host 1955223036ff. Site accessed 2 times.
```

25) Query the app using the Ansible uri module:

```
$ ansible app -m uri -a "url=http://172.18.0.3:5000 return_content=yes"
app | SUCCESS => {
    "changed": false,
    "content": "Hello from host 1955223036ff. Site accessed 3 times.\n",
    "content_length": "53",
    "content_type": "text/html; charset=utf-8",
    "cookies": {},
    "date": "Mon, 04 Sep 2017 06:35:34 GMT",
    "failed": false,
    "msg": "OK (53 bytes)",
    "redirected": false,
    "server": "Werkzeug/0.12.2 Python/2.7.5",
    "status": 200,
    "url": "http://172.18.0.3:5000"
}
```

26) Convert the ad hoc command just run into a task, and also add an additional task that uses the "fail" module to exit if the string "Hello from host" is not in the output captured by the uri module.

/labfiles/solutions/chap6/main.yaml.v9

Re-run the play and verify your output matched that shown below:

```
$ ansible-playbook main.yaml --limit app
. . . snip . . .
TASK [test app]
ok: [app]

TASK [fail]
skipping: [app]

PLAY RECAP
app                        : ok=9    changed=0    unreachable=0    failed=0
```

27) Convert the app deployment tasks into a block, and add a corresponding rescue section that will revert the app to a rollback version if the main block fails. Be sure to indent the existing tasks correctly to align under the new block:

File: main.yaml /labfiles/solutions/chap6/main.yaml.v10

```
  - hosts: app
    vars_files:
    - vars/app-version
    tasks:
    - name: copy rpms to host
      copy: src=files/app/ dest=/tmp
    - name: install Python
      yum: "name=/tmp/{{ item | basename }}"
      with_fileglob: files/app/*rpm
+   - name: App install with rollback
+     block:
→     - name: install requirements
→       pip: "name={{ item }}"
→       with_lines: cat "app/requirements.txt"
  . . . snip . . .
    # Indent existing tasks under new block #
  . . . snip . . .
+     - name: test app
+       uri: "url=http://172.18.0.3:5000 return_content=yes"
+       register: apptest
+     - fail:
+       when: '"Hello from host" not in apptest.content'
+     rescue:
+     - name: stop old app
+       systemd: name=app state=stopped
+     - name: install app version {{ rollback_version }}
+       copy:
+         src: app/app-v{{ rollback_version }}.py
+         dest: /root/
+     - name: link to app version {{ rollback_version }}
+       file:
+         state: link
+         dest: /root/app.py
+         src: /root/app-v{{ rollback_version }}.py
+     - name: start app
+       systemd: name=app state=started
```

28) Modify the included vars file incrementing the deploy version, and adding a rollback version:

File: vars/app-version

```
   ---
→  deploy_version: 1→2
+  rollback_version: 1
```

29) Create a broken version of the app by stripping the closing paren off line 4, then try running the play to verify that when the new version fails testing, the rollback version is automatically deployed:

```
$ sed '4 s/)//' app/app-v1.py > app/app-v2.py
$ diff app/app-v{1,2}.py
4c4
< app = Flask(__name__)
---
> app = Flask(__name__
$ ansible-playbook main.yaml --limit app
. . . snip . . .
TASK [install app version 2]
ok: [app]

TASK [install unit file]
ok: [app]

TASK [link to app version 2]
changed: [app]

TASK [start app]
ok: [app]

TASK [test app]
fatal: [app]: FAILED! => {"changed": false, "content": "", "failed": true,↵
   "msg": "Status code was not [200]: Request failed: <urlopen error [Errno↵
   111] Connection refused>", "redirected": false, "status": -1, "url":↵
   "http://172.18.0.3:5000"}

TASK [stop old app]
ok: [app]

TASK [install app version 1]
```

INTRODUCE UN ERROR EN EL FICHERO DE APP

```
ok: [app]

TASK [link to app version 1]
changed: [app]

TASK [start app]
changed: [app]

PLAY RECAP
app                        : ok=12   changed=3    unreachable=0    failed=1
```

30) Add two final tasks ~~(v10)~~ that always gather and print the service status regardless of if the new version deployed, or failed. Also, make it possible to call just the status tasks using a tag:

File: main.yaml /labfiles/solutions/chap6/main.yaml.v11

```
+    always:
+    - name: final app status
+      command: systemctl status app
+      register: appstatus
+      tags: [status]
+    - debug: "msg={{ appstatus.stdout_lines }}"
+      tags: [status]
```

31) Test the status functionality:

```
$ ansible-playbook main.yaml --limit app --tags status
TASK [final app status]
changed: [app]

TASK [debug]
ok: [app] => {
    "msg": [
        "- app.service - app1 daemon",
        "   Loaded: loaded (/etc/systemd/system/app.service; enabled; vendor preset: disabled)",
        "   Active: active (running) since Mon 2017-09-04 08:37:51 UTC; 2min 41s ago",
        " Main PID: 11045 (python)",
        "   CGroup: /docker/1955223036ffdd377aecec4b9ccee120412d5f742795431eedb29740f7b1d842/system.slice/app.service",
        "           |-11045 /usr/bin/python /root/app.py",
        "           `-11052 /usr/bin/python /root/app.py",
        "",
```

```
        "Sep 04 08:37:51 1955223036ff systemd[1]: Started app1 daemon.",
        "Sep 04 08:37:51 1955223036ff systemd[1]: Starting app1 daemon...",
        "Sep 04 08:37:51 1955223036ff python[11045]:  Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)",
        "Sep 04 08:37:51 1955223036ff python[11045]:  Restarting with stat",
        "Sep 04 08:37:51 1955223036ff python[11045]:  Debugger is active!",
        "Sep 04 08:37:51 1955223036ff python[11045]:  Debugger PIN: 304-188-081"
    ]
}

PLAY RECAP
app                        : ok=3    changed=1    unreachable=0    failed=0
```

Bonus: Scale out app servers

32) Change the deploy_version back to the working v1:

File: vars/app-version

```
---
deploy_version: 1
rollback_version: 1
```

33) Launch another base container to act as the new app host:

```
$ sh /labfiles/docker_centos_init.sh app2
```

34) Modify the playbook so that is matches all app hosts, and uses facts to determine the IP instead of a hard-corded value:

File: main.yaml /labfiles/solutions/chap6/main.yaml.v12

```
→ - hosts: app~app
+   gather_facts: No
    vars_files:
    - vars/app-version
    tasks:
+   - name: install iproute needed for fact gathering
+     yum: name=iproute
+   - setup:
  . . . snip . . .
      - name: test app
→       uri: "url=http://172.18.0.3{{ ansible_default_ipv4.address }}:5000 return_content=yes"
        register: apptest
```

35) Execute the entire playbook, and glory in a new app host being provisioned:

```
$ ansible-playbook main.yaml
. . . output omitted . . .
```

Cleanup

36) Remove the containers, and unset the config variable:

```
$ docker container rm -f redis app app2
redis
app
app2
$ unset ANSIBLE_CONFIG
```

Content

Chapter 7

ROLES

Roles

Ansible standard for organizing playbook content
Content organized into role structure is

- included automatically
- easier to share and re-use

Roles

As we have seen, Ansible allows you to organize the files it uses (playbooks, tasks, inventories, variable, etc.) pretty much any way you want. This flexibility is nice, but can make it difficult to share content within a community. Ansible 1.2 introduced the idea of roles as a standard for organizing content. When content is organized following the role standard, it automatically gets included in a way that makes high level playbooks more readable. It also makes it easier to share the content with others.

The standard structure for roles is as follows:

```
site.yaml
roles/rolename/
|-- defaults
|   `-- main.yaml
|-- files
|-- handlers
|   `-- main.yaml
|-- meta
|   `-- main.yaml
|-- tasks
|   `-- main.yaml
|-- templates
|-- tests
|   |-- inventory
|   `-- test.yaml
`-- vars
    `-- main.yaml
```

Roles are called within a playbook, as follows:

File: site.yaml

```
---
- hosts: class
  roles:
  - base
  - gl_ansible
```

The effect of this roles clause is essentially an automatic set of includes as follows:

- If roles/x/tasks/main.yml exists, tasks listed therein will be added to the play.
- If roles/x/handlers/main.yml exists, handlers listed therein will be added to the play.
- If roles/x/vars/main.yml exists, variables listed therein will be added to the play.
- If roles/x/defaults/main.yml exists, variables listed therein will be added to the play.
- If roles/x/meta/main.yml exists, any role dependencies listed therein will be added to the list of roles (1.3 and later).
- Any copy, script, template or include tasks (in the role) can reference files in roles/x/{files,templates,tasks}/ (dir depends on task) without having to path them relatively or absolutely.

Role Usage Details

Passing variables
Conditionals
Tags
Pre/Post tasks
Dependencies

Passing variables

Roles generally provide default values for any variables they use. If alternate values are wanted, these are usually set in the `vars/main.yml` within the role. If the role `var/` directory is missing, no error is generated. To pass variables when invoking a role use the following syntax:

File: `site.yaml`

```
---
- hosts: all
  roles:
  - webbase
  - { role: webvhost, docroot: 'app1', port: 8000 }
  - { role: webvhost, docroot: 'app2', port: 8001 }
```

Conditionals

To apply a conditional to every task in a role (actually evaluated for each task as if applied to them individually), it can be passed as follows:

File: `site.yaml`

```
---
- hosts: all
  roles:
  - { role: rhbase, when: "ansible_os_family == RedHat" }
```

Tags

If tags are passed on the role, then they override any tags that may be present within the role. Generally a better idea to split the role into several more focused roles instead:

File: `site.yaml`

```
---
- hosts: all
  roles:
  - { role: base, tags: ["setup"] }
  - { role: app1, tags: ["setup", "init"] }
  - { role: app1test, tags: ["test"] }
```

Pre/Post Tasks

Normally, if a play contains both role assignments, and bare tasks, all tasks in the role are executed first. To execute bare tasks before or after role, use the following syntax:

File: site.yaml

```
---
- hosts: all
  pre_tasks:
  - name: Get host ready
    script: files/prep.sh

  roles:
  - app42

  post_tasks:
  - name: Update infrastructure DB
    script: files/updatedb.sh
```

Dependencies

Roles can be stand alone, or may depend on other roles or initial system state. The role dependencies, if any, should be specified in the meta/main.yaml file for the role. When listing dependencies, variables can also be declared for the role; for example:

File: meta/main.yaml

```
---
dependencies:
  - mybase
  - { role: mydatacenter, dc: 2 }
```

The meta/main.yaml file also lists other meta-data used to organize the content within Ansible Galaxy. For example author, description, min_ansible_version, license, platforms suppported, categories, and tags.

QUIZ: Role Structure

Quiz and group discussion

Quiz: Role Structure

1. Which of the following are key benefits of Ansible roles?
   - Faster execution compared to normal playbooks.
   - Allows for automatic inclusion of content within playbook.
   - Enables sharing within a larger community.
   - Uses XML instead of YAML to define tasks.
2. Which file contains example variables and values that the role uses?
   - `vars/main.yaml`
   - `defaults/main.yaml`
   - `docs/main.yaml`
   - `readme.yaml`
3. Which of the following represent best practices with regards to roles?
   - Use tags to allow parts of a role to be executed independently.
   - Declare role dependancies in the `meta/main.yaml`.
   - Roles must NEVER depend on other roles.
   - Split complex roles into smaller roles to allow parts to be executed independently.
4. Which of the following can be contained within a role?
   - Tasks
   - Templates
   - Handlers
   - Variables
   - Sub-Roles

Creating Roles

Create structure
Define tasks, handlers
Document

- default vars
- requirements

Create Structure

Ansible does not require the full role directory/file structure to exist, and you can create directories by hand as you need them. Because it is easy, many use the **`ansible-galaxy init`** ***`role_name`*** command to build the initial structure. One advantage of this is that it produces template meta/main.yaml and README.md files that encourage the creation of documentation.

Define Content

The `tasks/main.yaml` file should contain a bare list of tasks. If the role does a single logical thing, then most likely all tasks will be in the `main.yaml` file. For readability and organization, you may place several task files in the `tasks/` directory and then use include statements (possibly with `when:` clauses) in the `main.yaml`. Remember that you should not manually include handlers, and vars as that will happen automatically.

Document

One of the primary reasons that roles exist is to encourage the sharing and re-use of playbook content. Documentation is key to effective sharing of roles. The declarative nature of Ansible modules coupled with the readability of YAML make most task lists self documenting. Use good `name:` clauses to make tasks even more readable (especially during execution). When using complex logic or data-structures, use YAML comments to make the operation more clear.

Since they are designed to be more portable, roles often feature more variables than playbooks developed for a single use case. The `defaults/main.yaml` file should contain a list of variables with safe/sane default values if possible. Again, comments can be a huge help to "outsiders" who attempt to use your role. If the role requires data where no default can reasonably be set, then it is common to put that variable in the `vars/main.yaml` file along with comments indicating the necessity of selecting a site appropriate value.

More formal documentation about the use of the module, is found in two primary files: `meta/main.yaml`, and `README.md`. The meta-data is primarily used by the Ansible Galaxy website and related commands to organize the content and expose basic info it. If the `README.md` exists, then it is rendered and displayed on the web landing page for the role on Galaxy.

Deploying Roles with Ansible Galaxy

Public (web) library of roles

- 1000's searchable by: platform, tag, role-type, author, keyword, etc.
- popularity judged by: stars, watchers, downlaods

`ansible-galaxy` → **CLI to manage roles**

Open Source, feel free to run your own Galaxy server

- pass `--server` or set in `ansible.cfg`

Ansible Galaxy

The `https://galaxy.ansible.com` website acts as a searchable community resource for roles. Hosting many thousands of roles, it is a great resource no only for downloading existing roles, but for "inspiration" when creating your own roles that solve similar problems. One of the primary ways to search for roles is by tag. Roles can be assigned up to 20 tags, which are arbitrary text labels.

Is this Role any Good?

Since anyone can post roles, and roles are not officially curated, supported, or endorsed, you must be the judge of a roles quality. Obviously anything that will be run against your systems should be reviewed and understood before use. Role popularity is measured in three ways and often is at least loosely correlated with role quality. Roles can be given stars by anyone with an account on the platform. Stars indicate the role is favorable to that user is some way. When a role is "watched", notifications are sent when the role is updated. The number of people watching a role is an indication of the amount of interest in the role and higher quality / more useful roles tend to be watched more often. The final and perhaps least useful metric is the number of downloads. Since some roles will naturally be used more often than more specialized roles, and since roles might be downloaded very frequently by certain systems (such as CI testing pipelines) the number of downloads should not be taken as being highly correlated with the quality of the role.

Role content is not actually stored on the ansible-galaxy site (which acts simply as a central index). Role content is generally stored in a GitHub repo.

ansible-galaxy Usage

The CLI for Ansible Galaxy can be used to manage roles on an Ansbile control host. It supports the following subcommands: delete, import, info, init, install, list, login, remove, search, setup. The following describes the function of each:

search ⇒ return list of up to first 1000 roles matching query terms.
info ⇒ Return all meta-data from Galaxy regarding the role.
init ⇒ Create new role structure.
install ⇒ Download role from Galaxy website (default), or configured site.
list ⇒ Names and version of all roles locally installed in roles_path.
remove ⇒ Remove role from local system.
delete ⇒ Remove role from Galaxy repo (requires login).
setup ⇒ Integrate with Travis build system.
login ⇒ Authenticate to Galaxy.
import ⇒ Download direct from GitHub repo (requires login).

DEMO: Deploying Roles with Ansible Galaxy

Explore roles in Ansible Galaxy
Deploy a LAMP stack using roles

Ansible Galaxy Website

Participate in a guided group review of the https://galaxy.ansible.com/ website.

Ansible Galaxy - Role Demo

By default Ansible tries to put roles into /etc/ansible/roles/, and if it can't write there, it puts them in ~/.ansible/roles/. Neither of these is convenient for us, so we install to our local roles directory:

```
[guru@station1]$ mkdir /home/guru/playbooks/roles
$ cd /home/guru/playbooks/roles
$ ansible-galaxy install --help
. . . output omitted . . .
```

File: roles.yml

```
+ - src: geerlingguy.mysql
+ - src: geerlingguy.apache
+ - src: geerlingguy.php
```

```
$ ansible-galaxy install -r roles.yml --roles-path=$(pwd)
- downloading role 'mysql', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-mysql/archive/2.9.4.tar.gz
- extracting geerlingguy.mysql to /home/guru/playbooks/roles/geerlingguy.mysql
- geerlingguy.mysql (2.9.4) was installed successfully
- downloading role 'apache', owned by geerlingguy
- downloading role from https://github.com/geerlingguy/ansible-role-apache/archive/3.0.3.tar.gz
- extracting geerlingguy.apache to /home/guru/playbooks/roles/geerlingguy.apache
- geerlingguy.apache (3.0.3) was installed successfully
- downloading role 'php', owned by geerlingguy
```

```
- downloading role from https://github.com/geerlingguy/ansible-role-php/archive/3.7.0.tar.gz
- extracting geerlingguy.php to /home/guru/playbooks/roles/geerlingguy.php
- geerlingguy.php (3.7.0) was installed successfully
```

Instructor examines, and discusses the basic contents of the roles: `README.md`, `meta/main.yml`, `defaults/main.yml`, `vars/mani.yml`, `tasks/main.yml`, and others as interest and time permit.

Now we create the playbook:

File: `lamp.yaml`

```
+ ---
+ - hosts: lamp
+   roles:
+   - geerlingguy.mysql
+   - geerlingguy.apache
+   - geerlingguy.php
```

Finally the container to test:

```
$ echo "[lamp]" > inventory
$ sh /labfiles/docker_centos_init.sh host1 | tee -a inventory
172.18.0.2  ansible_host=host1 ansible_connection=docker
$ ansible-playbook -i inventory lamp.yaml
. . . snip . . .
TASK [geerlingguy.php : Ensure PHP packages are installed.] *********************
fatal: [172.18.0.2]: FAILED! => {"changed": false, "msg": "No package matching ➥
      'php-imap' found available, installed or updated", "rc": 126, "results": ["No package matching 'php-imap' ➥
      found available, installed or updated"]}
PLAY RECAP *********************************************************************
172.18.0.2                 : ok=55   changed=10   unreachable=0    failed=1
```

Third role fails because one of the PHP modules is not available in the repos currently configured. As a workaround, comment out that module in the `roles/geerlingguy.php/vars/RedHat.yml` file, then try running again.

```
$ ansible-playbook -i inventory lamp.yaml
. . . snip . . .
TASK [geerlingguy.php : Ensure PHP packages are installed.] **********************
fatal: [172.20.0.2]: FAILED! => {"changed": false, "msg": "No package matching ➥
      'php-opcache' found available, installed or updated", "rc": 126, "results": ["No package matching 'php-opcache'➥
      found available, installed or updated"]}
        to retry, use: --limit @/home/guru/playbooks/roles/lamp.retry
```

The roles make good use of checks and quickly advance to the PHP installation playbook, but fail on another module. Comment out the `php-opcache`, and `php-pecl-apcu` modules as well (also both not available from current repos). Then run one final time:

```
$ ansible-playbook -i inventory lamp.yaml
. . . snip . . .
PLAY RECAP *********************************************************************
172.20.0.2                 : ok=68   changed=5    unreachable=0    failed=0
```

All done! Time to test:

```
$ HEAD 172.18.0.2
403 Forbidden
Connection: close
Date: Fri, 18 Jan 2019 01:18:45 GMT
Accept-Ranges: bytes
ETag: "1321-5058a1e728280"
Server: Apache/2.4.6 (CentOS) OpenSSL/1.0.2k-fips PHP/5.4.16
Content-Length: 4897
Content-Type: text/html; charset=UTF-8
Last-Modified: Thu, 16 Oct 2014 13:20:58 GMT
Client-Date: Fri, 18 Jan 2019 01:18:45 GMT
Client-Peer: 172.20.0.2:80
Client-Response-Num: 1
```

```
$ docker container exec host1 ps -ef
UID        PID  PPID  C STIME TTY      TIME CMD
root         1     0  0 02:30 ?   00:00:00 /sbin/init
root        16     1  0 02:30 ?   00:00:00 /usr/lib/systemd/systemd-journald
dbus       461     1  0 02:31 ?   00:00:00 /bin/dbus-daemon --system --address=systemd: --nofork --nopidfile --systemd-activation 00:00:00 /bin/sh
/usr/bin/mysqld_safe --basedir=/usr
mysql      522     1  0 02:31 ?   00:00:01 /usr/libexec/mysqld --basedir=/usr --datadir=/var/lib/mysql --plugin-dir=/usr/lib64/mysql/ 00:00:00
mysql     1053   522  0 02:31 ?   /usr/sbin/httpd -DFOREGROUND
root      4084     1  0 02:44 ?   00:00:00     /usr/sbin/httpd     -DFOREGROUND
apache    4085  4084  0 02:44 ?   00:00:00     /usr/sbin/httpd     -DFOREGROUND
apache    4086  4084  0 02:44 ?   00:00:00     /usr/sbin/httpd     -DFOREGROUND
apache    4087  4084  0 02:44 ?   00:00:00     /usr/sbin/httpd     -DFOREGROUND
apache    4088  4084  0 02:44 ?   00:00:00     /usr/sbin/httpd     -DFOREGROUND
apache    4089  4084  0 02:44 ?   00:00:00 ps -ef
root      4096     0  0 02:46 ?
```

Lab 7

Estimated Time:
R7: 70 minutes

Task 1: Converting Playbooks to Roles [R7]

Page: 7-12 Time: 20 minutes

Requirements: 💻 (1 station)

Task 2: Creating Roles from Scratch [R7]

Page: 7-21 Time: 30 minutes

Requirements: 💻 (1 station)

Task 3: Ansible Galaxy Roles [R7]

Page: 7-25 Time: 20 minutes

Requirements: 💻 (1 station)

Objectives

- Create the standard directory and file structure to hold a role
- Convert a playbook into a role

Requirements

- (1 station)

Relevance

Lab 7
Task 1
Converting Playbooks to Roles [R7]

Estimated Time: 20 minutes

1) The Instructor must ensure that the sym link needed by this lab has been created on the shared classroom server:

```
[server1]# [ $(hostname) == server1.example.com ] && ln -s /export/netinstall/CENTOS /var/www/html/centos7
```

CREAR EL ENLACE EN EL SERVER1 PARA QUE PUEDA USAR ESE REPOSITORIO DE INSTALACIÓN LAS ESTACIONES

2) Create a directory to hold the roles, and install the `tree` command:

```
$ mkdir ~/playbooks/roles
$ cd ~/playbooks
$ sudo yum install -y tree
. . . output omitted . . .
```

3) Create the start of a playbook that creates the basic directory and file structure needed for a role:

File: ~/playbooks/makerole.yaml /labfiles/solutions/chap7/makerole.yaml

```
+ ---
+ - hosts: localhost
+   gather_facts: No
+   vars:
+     dirs:
+     - defaults
+     - handlers
+     - meta
+     - tasks
+     - vars
+     - files
+     - templates
+   vars_prompt:
+   - name: "name"
+     prompt: "Role name"
+     private: No
```

4) Add a task that loops over the dirs var and creates a directory for each:

File: ~/playbooks/makerole.yaml

```
+   tasks:
+   - name: create directories
+     file:
+       state: directory
+       path: "{{ name }}/{{ item }}"
+     loop: "{{ dirs }}"
```

5) Add a task that creates a base YAML file with a comment:

File: ~/playbooks/makerole.yaml

```
+   - name: create base yaml
+     copy:
+       content: |
+         ---
+         #{{ item.1 }} file for {{ name }}
+       dest: "{{ name }}/{{ item.1 }}/main.yaml"
+     with_indexed_items: "{{ dirs }}"
+     when: item.0|int < 5
```

Note the use of the relatively rare `with_indexed_items` to access the indexes and then only create files for the first 5 indexed items in the var.

6) Execute the playbook passing a value at the prompt and examine the output:

```
$ cd ~/playbooks/roles
$ ansible-playbook ../makerole.yaml
Role name: vhost
PLAY [localhost]

TASK [create directories]
ok: [localhost] => (item=defaults)
ok: [localhost] => (item=handlers)
. . . snip . . .

PLAY RECAP
localhost                  : ok=4    changed=0    unreachable=0    failed=0
$ tree vhost/
vhost/
|-- defaults
|   `-- main.yaml
|-- files
|-- handlers
|   `-- main.yaml
|-- meta
|   `-- main.yaml
|-- tasks
|   `-- main.yaml
|-- templates
`-- vars
    `-- main.yaml
```

```
8 directories, 7 files
```

This is the base structure needed by roles. A few other directories and files and common, but not necessary.

7) Unpack a traditional playbook (and related files) into the directory, and examine the playbook until you are comfortable you understand what it does:

```
$ cd vhost
$ tar xf /labfiles/web_vhost_playbook.tar
$ less web.yaml
. . . output omitted . . .
```

8) Create a new host (container) to manage, and verify Ansible can connect:

```
$ sh /labfiles/docker_centos_init.sh web
$ export ANSIBLE_CONFIG=~/docker/ansible.cfg
$ ansible web -m ping
web | SUCCESS => {
    "changed": false,
    "failed": false,
    "ping": "pong"
}
```

9) Run the playbook, and verify the sites are functioning:

```
$ ansible-playbook web.yaml
. . . snip . . .
TASK [Configure vhost]
changed: [web] => (item=site1)
changed: [web] => (item=site2)

RUNNING HANDLER [restart apache]
changed: [web]

PLAY RECAP
web                        : ok=11   changed=9    unreachable=0    failed=0
$ ansible web -m uri -a "url=http://site1 return_content=true"
web | SUCCESS => {
    "accept_ranges": "bytes",
    "changed": false,
```

```
    "connection": "close",
    "content": "<html>Welcome to site1 website.</html>",
. . . snip . . .
    "status": 200,
    "url": "http://site1"
}
$ ansible web -m uri -a "url=http://site2 return_content=true"
web | SUCCESS => {
    "accept_ranges": "bytes",
    "changed": false,
    "connection": "close",
    "content": "<html>Welcome to site2 website.</html>",
. . . snip . . .
    "status": 200,
    "url": "http://site2"
```

Converting a Playbook into a Role

10) Start by moving the port variable into the `defaults/main.yml`:

File: `defaults/main.yaml`

```
   ---
   #defaults file for roles/vhost
+  port: 80
```

11) Move vhost variable into the `vars/main.yml`:

File: `vars/main.yml`

```
   ---
   #vars file for roles/vhost
+  vhosts:
+  - site1
+  - site2
```

12) Move (copy and paste?) the core setup tasks into the `tasks/install.yaml`:

File: tasks/install.yaml

```
+ ---
+ #tasks file for roles/vhost
+ - name: Install base repo
+   yum_repository:
+     name: centos7
+     description: centos7 repo
+     baseurl: http://10.100.0.254/centos7
+     gpgkey: http://10.100.0.254/centos7/RPM-GPG-KEY-CentOS-7

+ - name: install iproute
+   yum: name=iproute

+ - name: Gather facts
+   setup:

+ - name: Add sites to hosts file
+   lineinfile:
+     dest: /etc/hosts
+     line: "{{ ansible_default_ipv4.address }} {{ item }}"
+     unsafe_writes: true
+   with_items: "{{ vhosts }}"

+ - name: Install Apache
+   yum: name=httpd
```

13) Move the remaining setup tasks into the tasks/setup.yaml:

File: tasks/setup.yaml

```
+ - name: Configure web for alternate port
+   lineinfile:
+     regexp: ^Listen
+     line: "Listen {{ port }}"
+     dest: /etc/httpd/conf/httpd.conf
+   notify: restart apache

+ - name: Activate name based routing
+   lineinfile:
+     dest: /etc/httpd/conf/httpd.conf
+     line: "NameVirtualHost {{ ansible_default_ipv4.address }}"
+   notify: restart apache

+ - name: Create virt
+   file: dest=/var/www/html/virt/{{ item }} state=directory
+   with_items: "{{ vhosts }}"

+ - name: Index.html
+   copy:
+     dest: /var/www/html/virt/{{ item }}/index.html
+     content: "<html>Welcome to {{ item }} website.</html>"
+   with_items: "{{ vhosts }}"

+  - name: Copy secret data
+    copy:
+      src: data
+      dest: /var/www/html/virt/{{ item }}/index.html
+  with_items: "{{ vhosts }}"

+ - name: Configure vhost
+   template: src=vhost.j2 dest=/etc/httpd/conf.d/vhost-{{ item }}.conf
+   with_items: "{{ vhosts }}"
+   notify: restart apache
```

14) Add include statements to the `main.yaml` to pull in the other tasks:

File: tasks/main.yaml

```
   ---
   #tasks file for roles/vhost
+  - include: install.yaml
+  - include: setup.yaml
```

15) Move the handler into the handlers/main.yaml:

File: handlers/main.yaml

```
   ---
   #handlers file for roles/vhost
+  - name: restart apache
+    service: name=httpd state=restarted
```

16) Simply move the template and file into their respective directories, no editing required:

```
$ mv vhost.j2 templates/
$ mv data files/
```

17) Create a new site-web.yaml that maps the host to the role:

File: ~/playbooks/roles/site-web.yaml

```
+  ---
+  - hosts: ~web
+    roles:
+    - vhost
```

18) Try executing the new role-based playbook against the existing web host (which should report no changes since it is already configured), then again passing a different port to override the role default:

```
$ ansible-playbook site-web.yaml
. . . snip . . .
PLAY RECAP
web                        : ok=13   changed=5    unreachable=0    failed=0
$ ansible-playbook -e port=8000 site-web.yaml
```

```
. . . snip . . .
PLAY RECAP
web                        : ok=13   changed=5    unreachable=0    failed=0
$ ansible web -m uri -a "url=http://site2:8000 return_content=true"
web | SUCCESS => {
. . . snip . . .
    "status": 200,
    "url": "http://site2:8000"
}
```

Nice job! You have successfully converted an existing playbook into a more portable role.

Cleanup

19) Remove the container:

```
$ docker container rm -f web
```

Objectives

- Create a simple role
- Use assertions and facts to perform test against a machine's state

Requirements

- (1 station)

Relevance

Lab 7
Task 2
Creating Roles from Scratch [R7]

Estimated Time: 30 minutes

1) Create a new role directory to hold a role you will create from scratch:

```
$ mkdir -p ~/playbooks/roles/spec/{defaults,vars,tasks}
$ cd  ~/playbooks/roles/spec
```

2) Create tasks that implement checks against a server and fail if it doesn't meet the desired specification defined by variables. For example, a spec test for memory might look like:

File: spec/defaults/main.yml

```
---
# defaults file for spec
# minimum memory in Gigabytes
mem: 1
```

File: spec/vars/main.yml

```
---
# vars file for spec
mem: 8
```

File: spec/tasks/main.yml

```
--
# tasks file for spec
- include: mem.yaml
```

File: spec/tasks/mem.yaml

```
- name: Memory check
  assert:
    that:
    - ansible_memtotal_mb >= mem * 1024
    msg: "Memory is {{ ansible_memtotal_mb }}mb, but required {{ mem * 1024 }}mb"
```

3) Create ~/playbooks/roles/test.taml that binds the spec role to localhost. When testing, choose default variable values that cause the tests to pass and then override in the vars file (or with a value passed via CLI -e option) with a value that causes the test to fail.

Example output with mem: 1 in spec/default/main.yaml and nothing in spec/vars/main.yaml:

```
$ ansible-playbook test.yml
. . . snip . . .
TASK [spec : Memory minimum check]
ok: [spec] => {
    "changed": false,
    "failed": false,
    "msg": "All assertions passed"
}
```

Example output with mem: 8 in spec/vars/main.yaml:

```
$ ansible-playbook test.yml
. . . snip . . .
TASK [spec : Memory minimum check]
fatal: [spec]: FAILED! => {
    "assertion": "ansible_memtotal_mb >= mem * 1024",
    "changed": false,
    "evaluated_to": false,
    "failed": true,
    "msg": "Memory is 3951mb, but required 8192mb"
}
```

Example output passing vars via CLI instead:

```
$ ansible-playbook -e "{mem: 1, net_iface_num: 4, ping_list: ['127.0.0.1', '10.100.0.253']}" test.yaml

PLAY [localhost] ***************************************************************
```

```
TASK [Gathering Facts] *********************************************************
ok: [localhost]

TASK [spec : Memory check] *****************************************************
ok: [localhost] => {
    "changed": false,
    "msg": "All assertions passed"
}

TASK [spec : Network interface num check] **************************************
ok: [localhost] => {
    "changed": false,
    "msg": "All assertions passed"
}

TASK [spec : Ping test] ********************************************************
ok: [localhost] => (item=127.0.0.1)
failed: [localhost] (item=10.100.0.253) => {"changed": false, "cmd": ["ping",
"-c1", "10.100.0.253"], "delta": "0:00:03.008420", "end": "2019-01-18
13:34:24.657606", "item": "10.100.0.253", "msg": "non-zero return code", "rc":
1, "start": "2019-01-18 13:34:21.649186", "stderr": "", "stderr_lines": [],
"stdout": "PING 10.100.0.253 (10.100.0.253) 56(84) bytes of data.\nFrom
10.100.0.2 icmp_seq=1 Destination Host Unreachable\n\n--- 10.100.0.253 ping
statistics ---\n1 packets transmitted, 0 received, +1 errors, 100% packet loss,
time 0ms", "stdout_lines": ["PING 10.100.0.253 (10.100.0.253) 56(84) bytes of
data.", "From 10.100.0.2 icmp_seq=1 Destination Host Unreachable", "", "---
10.100.0.253 ping statistics ---", "1 packets transmitted, 0 received, +1
errors, 100% packet loss, time 0ms"]}
        to retry, use: --limit @/home/guru/playbooks/roles/test.retry

PLAY RECAP *********************************************************************
localhost                  : ok=3    changed=0    unreachable=0    failed=1
```

4) Extend the role to check for at least 4 of the following:

- Number of network interfaces matches specified value.
- Number of CPUs matches specified value.
- At least specified amount of disk space free on root filesystem.
- Can ping the provided list of hosts.
- Has the specified user account.
- Can connect to the specified URL.

- Matches the specified Linux distribution and major version.
- Matches the specified kernel version.

Test your tasks by setting variables and running.

Objectives
- Use Ansible Galaxy to manage roles

Requirements
- (1 station)

Relevance

Lab 7
Task 3
Ansible Galaxy Roles [R7]

Estimated Time: 20 minutes

1) Install a popular role from Ansible Galaxy that can deploy Redis:

```
$ ansible-galaxy search redis
Found 321 roles matching your search:

 Name                                          Description
 ----                                          -----------
 jpnewman.redis                                Redis
 kunik.redis                                   install redis
 hostclick.redis                               Ansible redis role
. . . snip . . .
$ cd ~/playbooks/roles
$ ansible-galaxy install --roles-path . DavidWittman.redis
- downloading role 'redis', owned by DavidWittman
- downloading role from https://github.com/DavidWittman/ansible-redis/archive/1.2.5.tar.gz
- extracting DavidWittman.redis to /home/guru/playbooks/roles/DavidWittman.redis
- DavidWittman.redis (1.2.5) was installed successfully
```

Note that the search does not take into account any download count, or any other metric, to try and judge how "good" a role is. If fact, many of the roles listed have almost nothing to do with Redis other than requiring it as a dependency.

2) Take a look at the files for the role:

```
$ find DavidWittman.redis/
. . . output omitted . . .
$ less DavidWittman.redis/README.md
. . . output omitted . . .
```

Many simple roles can be immediately understood by just looking at the `tasks/main.yaml` and perhaps the `vars/main.yaml`. For more complicated roles, the `README.md` is the best place to start.

3) Create a simple playbook that applies the role to a single host and execute to test:

File: ~/playbooks/roles/redis.yaml

```
+ ---
+ - hosts: redis01
+   roles:
+     - DavidWittman.redis
```

```
$ sh /labfiles/docker_centos_init.sh redis01
$ ansible-playbook redis.yaml
. . . output omitted . . .
$ redis-cli -h 172.18.0.2 ping
PONG
$ docker container rm -f redis01
redis01
```

4) Now try using the same role to build a fully redundant / HA cluster of Redis instances. First create the containers (note that in the "real" world, these containers would be spread across many hosts, or simply be real physical hosts:

```
$ for i in redis-{master,{slave,sentinel}0{1,2,3}}; do
> sh /labfiles/docker_ubuntu_init.sh $i | tee -a redis_cluster_inventory2; done
redis-master ansible_host=172.18.0.2
redis-slave01 ansible_host=172.18.0.3
redis-slave02 ansible_host=172.18.0.4
redis-slave03 ansible_host=172.18.0.5
redis-sentinel01 ansible_host=172.18.0.6
redis-sentinel02 ansible_host=172.18.0.7
redis-sentinel03 ansible_host=172.18.0.8
```

5) Create an inventory file so that the hosts are assigned to groups. Also, note the use of a hostvar that the role looks for as a way of determining which hosts will act as sentinels:

File: redis_cluster_inventory

```
+ [redis-master]
+ redis-master

+ [redis-slave]
+ redis-slave0[1:3]

+ [redis-sentinel]
+ redis-sentinel0[1:3] redis_sentinel=True
```

6) Finally, create a playbook that maps the inventory groups to roles, and sets needed vars:

File: ~/playbooks/roles/redis_cluster.yaml

```
+ ---
+ - name: configure the master redis server
+   hosts: redis-master
+   roles:
+     - DavidWittman.redis
+
+ - name: configure redis slaves
+   hosts: redis-slave
+   vars:
+     - redis_slaveof: redis-master 6379
+   roles:
+     - DavidWittman.redis
+
+ - name: configure redis sentinel nodes
+   hosts: redis-sentinel
+   vars:
+     - redis_sentinel_monitors:
+       - name: master01
+         host: redis-master
+         port: 6379
+   roles:
+     - DavidWittman.redis
```

7) Kick off the play and watch as the magic unfolds... or, go take a break :)

```
$ time ansible-playbook -i redis_cluster_inventory redis_cluster.yaml
. . . output omitted . . .
```

8) Test that the cluster is happy by connecting to one of the sentinels and querying:

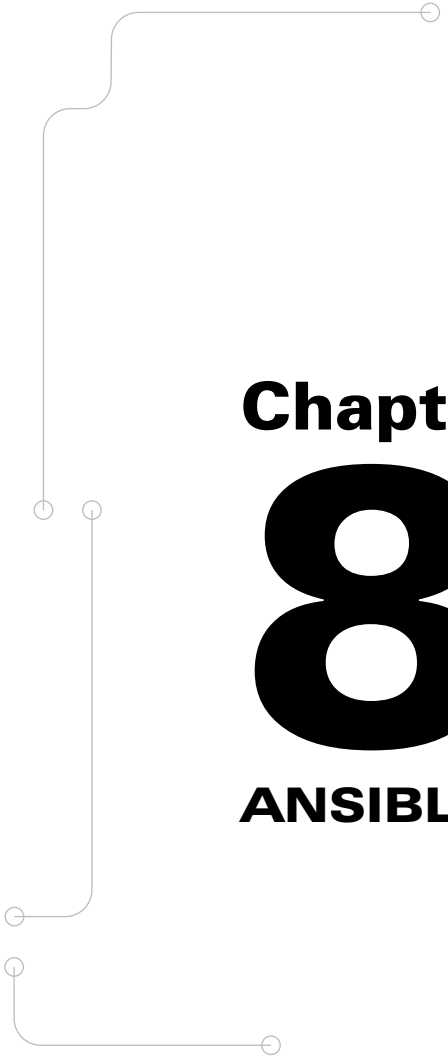
```
$ redis-cli -h 172.18.0.6 -p 26379 sentinel masters
1)  1) "name"
    2) "master01"
    3) "ip"
    4) "172.18.0.2"
. . . snip . . .
$ redis-cli -h 172.18.0.6 -p 26379 sentinel slaves master01 | grep -A3 master-link-status
master-link-status
ok
master-host
172.18.0.2
--
master-link-status
ok
master-host
172.18.0.2
--
master-link-status
ok
master-host
172.18.0.2
```

Cleanup

9) Remove the containers:

```
$ docker container rm -f $(docker ps -qa)
```

Content



Chapter 8

ANSIBLE VAULT

Configuring Ansible Vault

ansible-vault

- create → new encrypted file
- encrypt_string → STDIN/STDOUT or arg
- {en,de}crypt → file
- view → decrypt to STDOUT
- edit → decrypt, edit, encrypt
- rekey → decrypt, encrypt with new password

Dealing with Sensitive Data

One of the advantages of being able to describe the state of infrastructure in code is that it can be checked into a source control system, versioned, tracked, rolled back, etc. A challenge faced when checking playbooks (and their corresponding var files, inventories, and other related files) is that they often contain sensitive data such as passwords, and keys. Some source control systems offer methods of protecting this type of data, but all too often the data ends up included in a normal repo and accessible to the wrong set of people.

Ansible ships with a binary called **ansible-vault** that can be used to encrypt any sensitive data used with Ansible. The primary advantage of vault is that it has better integration with the other Ansbile commands. For example, when Ansible encounters a vault encrypted block, it can automatically prompt for the password to decrypt it, and then continue processing. The following show basic usage to encrypt a string:

```
$ ansible-vault encrypt_string "My secret string"
New Vault password: magic Enter
Confirm New Vault password: magic Enter
!vault |
          $ANSIBLE_VAULT;1.1;AES256
          61656532353462336636353839633864633063396536303935633563613837303139376630633666306564353137336264303836653762356239616638633464663838346664343863333530376162353130a656665613839376463346331373633313566656235303534376437333935373535
```

Encryption successful

Vault requires a password to encrypt the data, and it can be provided by one of two methods: "ask-vault-pass" or "vault-password-file". Without either of these options present, vault will default to "ask-vault-pass".

Usage: ansible-vault [create|decrypt|edit|encrypt|encrypt_string|rekey|view] [--help] [options] [vaultfile.yml]

Create

The create option will open a temporary file in your editor (determined by $EDITOR environment variable). After saving your changes to the temporary file, it will be saved as the filename provided on the command line; for example:

```
$ ansible-vault create vault_vars.yaml
New Vault Password: magic Enter
Confirm New Vault Password: magic Enter
. . . output omitted (File opens in your EDITOR.) . . .
(File contents): status: "Success"

$ cat vault_vars.yaml
$ANSIBLE_VAULT;1.1;AES256
62323564633003831633161646432353536323832623131323261303820a30653862666164366265623137366162346235613506161323735306564636230396133346637353663396296136306534303831373637303564336436
```

```
30623234373753630663832383863343 6
```

To undo any encryption, be sure you remember which password was used to encrypt your file!

```
$ ansible-vault decrypt vault_vars.yaml
Vault password: magic Enter
Decryption successful
$ cat vault_vars.yaml
status: "Success"
```

Edit

Rather than continuously running "encrypt" and "decrypt", Ansible provides a way to directly edit encrypted data using the "edit" argument. Ansible will then use a similar approach to "create" by creating a temporary file, loading the decrypted file content, then saving the changes as encrypted data. The "edit" feature only works on already-encrypted data. We can change any existing file to an encrypted file by using "ansible-vault encrypt":

```
$ ansible-vault edit not_yet_encrypted.yaml
ERROR! input is not vault encrypted data for /root/testing/vault
$ ansible-vault edit encryped.yaml
Vault password: magic Enter
. . . output omitted (File opens in your EDITOR successfully.) . . .
```

Encrypt

```
$ ansible-vault encrypt test.yaml
New Vault password: magic Enter
Confirm New Vault password: magic Enter
Encryption successful
```

Rekey

To change the password on one or more files.

```
$ ansible-vault rekey test.yaml
Vault password: magic Enter
New Vault password: newmagic Enter
Confirm New Vault password: newmagic Enter
Rekey successful
```

Password File

Prior to Ansible 2.4, The password used with vault must be the same for all files you wish to use together at the same time. After 2.4, multiple passwords can be passed using the "vault-id" option. A good way to ensure each file is using the same password is to use the `vault_password_file` option in your `ansible.cfg`. Any file can be used as a "password file". For example:

File: `password.file`

```
This is a password.
```

NOTE: The password file can't be empty! Otherwise, you will see this message:

```
ERROR! Unexpected Exception, this is probably a bug: key_material
```

The easiest way to use a password file is to edit a local `ansible.cfg` to include the path to the vault password file:

File: `./ansible.cfg`

```
+ [defaults]
+ vault_password_file=./password.file
```

With/`test.yaml`these in place, vault will use the `password.file` contents as the

password for each ansible-vault command run in this directory.

Vault IDs

Pre 2.4 only a single password possible
Vault protocol 1.2 supports labels
Ansible execution commands now allow specifying multiple vaults

- passwords can come from file or prompt

Vault-IDs

Starting in version 2.4, Ansible supports the "vault id" concept. Essentially this is a way to add a label to the password, or password file used with vault-protected content; for example:

```
$ ansible-vault --vault-id label@prompt create new.yaml
New vault password (label):
Confirm vew vault password (label):
```

(Yes, there is a typo in this feature as of 9/9/2017)

The @prompt will ask for user input on STDIN. If any other value is present, it will look for a file by that name.

The "label" portion populates the prompt message. This can be used as a reminder of which password was used to encrypt the file. It is not required, but if provided, it is stored in the header of the file in clear text.

```
$ ansible-vault --vault-id dev@passfile create demo.yaml
$ ansible-vault --vault-id passfile view demo.yaml
Content
```

The –vault-id can be used in lieu of the `--vault-password-file` or `--ask-vault-pass` options, or it can be used in combination with them. When using **`ansible-vault`** commands that encrypt content only one vault-id can be used.

Executing with Ansible Vault

Pre 2.4
- one vault password
- `--ask-vault-pass`
- `--vault-password-file=`

2.4+
- multiple vaults
- `--vault-id @prompt`
- `--vault-id @`*`filename`*

Use `{host,group}_vars` **directories**

Playbook Execution and Vault

When dealing with encrypted assets, Ansible needs access to the password(s) on execution. Password can be provided interactively, or can come from a file; for example:

```
$ ansible-playbook site.yml
ERROR: A vault password must be specified to decrypt vars/vault.yml
$ ansible-playbook --ask-vault-pass site.yaml
Vault password: magic Enter
. . . output omitted . . .
$ ansible-playbook --vault-password-file=vars_vault_pass site.yaml
. . . output omitted . . .
```

The location of the vault password file can be set in the config defaults section using `vault_password_file=`, or withthe `$ANSIBLE_VAULT_PASSWORD_FILE` environment variable.

Recommended Practice Vault Variables

As always, striking the correct balance between security and ease of use is critical. Encrypting playbook data makes it harder to edit, harder to track in version control, slower to use, etc. Avoid overdoing things and encrypting more than is needed. If you really have a case where even revealing the names of the variables or task details is unacceptable, then go ahead and encrypt the task file, but this is almost always overkill.

Using `host_vars/` and `group_vars/` directories and splitting encrypted files out into separate files is a great pattern. For example:

```
playbook_root/
├── group_vars
│   └── web
│       ├── vars.yaml
│       └── vault.yaml
├── host_vars
│   └── master.example.com
│       ├── vars.yaml
│       └── vault.yaml
└── site.yaml
```

Speeding up Encryption

If you are encrypting a lot of data, then the default library used can become a noticeable bottleneck. Per the docs:

By default, Ansible uses PyCrypto to encrypt and decrypt vault files. If you have many encrypted files, decrypting them at startup may cause a perceptible delay. To speed this up, install the cryptography package: "`pip install cryptography`"

DEMO: Configuring Ansible Vault

Using vault-ids to support multiple environments
Creating an encrypted playbook

dev/prod Vaults

First we create different vault encrypted variable files with values appropriate for each environment:

```
$ ansible-vault --vault-id=dev@prompt create dev.yaml
New vault password (dev): dev pass Enter
Confirm new vault password (dev): dev pass Enter
i dbpass: secretdevdbpass Esc : w q Enter
$ cat dev.yaml
$ANSIBLE_VAULT;1.2;AES256;dev
613636613866636632393633613630303639656236616233 61
333536303062633366346239396631656539316632306235 31
. . . snip . . .
$ ansible-vault --vault-id=prod@prompt create prod.yaml
New vault password (prod): prod pass Enter
Confirm new vault password (prod): prod pass Enter
i dbpass: secretproddbpass Esc : w q Enter
```

To show the use of storing vault passwords in files instead, we put the variables common to both environments in a vault that uses a password file:

```
$ echo "common pass" > common.vaultpass
$ ansible-vault --vault-id=common@common.vaultpass create vault_common.yaml
i dbname: app1
dbuser: dbuser1 Esc : w q Enter
```

Now, a simple playbook that can display the value of the variables loaded:

File: vault_demo.yaml

```
+ ---
+ - hosts: localhost
+   vars_files:
+   - vault_common.yaml
+   tasks:
+   - debug:
+       var: "{{ item }}"
+     loop:
+     - dbname
+     - dbuser
+     - dbpass
```

Let's try first executing the playbook with just the common vault (decrypted with the password from the file):

```
$ ansible-playbook --vault-id @common.vaultpass vault_demo.yaml
TASK [debug]
ok: [localhost] => (item=dbname) => {
    "dbname": "app1",
    "item": "dbname"
}
ok: [localhost] => (item=dbuser) => {
    "dbuser": "dbuser1",
    "item": "dbuser"
}
ok: [localhost] => (item=dbpass) => {
    "dbpass": "VARIABLE IS NOT DEFINED!",
    "item": "dbpass"
}
```

Now specifying two vault-ids:

```
$ ansible-playbook --vault-id @common.vaultpass --vault-id dev@prompt vault_demo.yaml
Vault password (dev): dev pass Enter
. . . snip . . .
ok: [localhost] => (item=dbpass) => {
    "dbpass": "VARIABLE IS NOT DEFINED!",
    "item": "dbpass"
}
```

Why is the variable still undefined?

We tell the play to include the vars in the dev file, and execute again:

```
$ ansible-playbook --vault-id @common.vaultpass --vault-id dev@prompt -e @dev.yaml vault_demo.yaml
Vault password (dev): dev pass Enter
. . . snip . . .
ok: [localhost] => (item=dbpass) => {
    "dbpass": "secretdevdbpass",
    "item": "dbpass"
}
$ ansible-playbook --vault-id @common.vaultpass --vault-id prod@prompt -e @prod.yaml vault_demo.yaml
Vault password (prod): prod pass Enter
. . . snip . . .
ok: [localhost] => (item=dbpass) => {
    "dbpass": "prodsecretdbpass",
    "item": "dbpass"
}
```

Encrypting an entire playbook

Remember that you can encrypt anything Ansible uses with vault. How about encrypting the entire playbook with the common vault password? (AKA how to annoy your co-workers, and make your source control system diff functions useless...)

```
$ ansible-vault encrypt --vault-id @common.vaultpass vault_demo.yaml
Encryption successful
$ cat vault_demo.yaml
$ANSIBLE_VAULT;1.1;AES256
36656636663366237373632316539626437323866366
65376634383736461623938613064316562323137 65
31646230643966538653638633263393465653038 31
37316531306263239393538616338646664303463 61
. . . snip . . .
$ ansible-playbook --vault-id @common.vaultpass --vault-id prod@prompt -e @prod.yaml vault_demo.yaml
Vault password (prod): prod pass Enter
. . . output omitted . . .
```

Lab 8

Estimated Time:
R7: 15 minutes

Task 1: Ansible Vault [R7]

Page: 8-10 Time: 15 minutes

Requirements: 💻 (1 station)

Objectives

- Use Ansible Vault to store sensitive data.

Requirements

- (1 station)

Relevance

Lab 8
Task 1
Ansible Vault [R7]

Estimated Time: 15 minutes

1) Create a file to store a vault password:

```
$ mkdir ~/vault; cd ~/vault
$ echo "mysecret" > passfile
```

2) Create a local `ansible.cfg` file to use this file as a password:

File: ~/vault/ansible.cfg

```
+ [defaults]
+ vault_password_file=~/vault/passfile
```

3) Create an encrypted vars file with a couple of variables:

```
$ ansible-vault create ~/vault/vars.yaml
i greeting: "Hello World" Esc : w q Enter
```

4) Verify the information was encrypted:

```
$ cat vars.yaml
$ANSIBLE_VAULT;1.1;AES256
623332303034316232370306239306534626537313065393733643730
363033333235373530363383 90a3863623863343535383030316433 31
373865646130376235614316635393535326466313766356233393530
383030636135383366 0a4616565323039323836396662333038663836
623361396466313634621326334353633623961356239653930616264
326263353131373132345353039 31

(Output will vary)
```

5) Access the encrypted variable in a simple ad hoc command:

```
$ ansible localhost -e @vars.yaml -m debug -a 'var=greeting'
localhost | SUCCESS => {
    "msg": "Hello World"
}
```

6) Create two files, one will be used to store all unencrypted variables, the other to store only sensitive variables (encrypted):

File: my_vars.yaml

```
+ my_name: "Bob"
+ my_age: "{{ vault_age }}"
+ my_weight: "{{ vault_weight }}"
+ my_eye_color: "Blue"
```

7) Create the vault vars file:

File: vault_vars.yaml

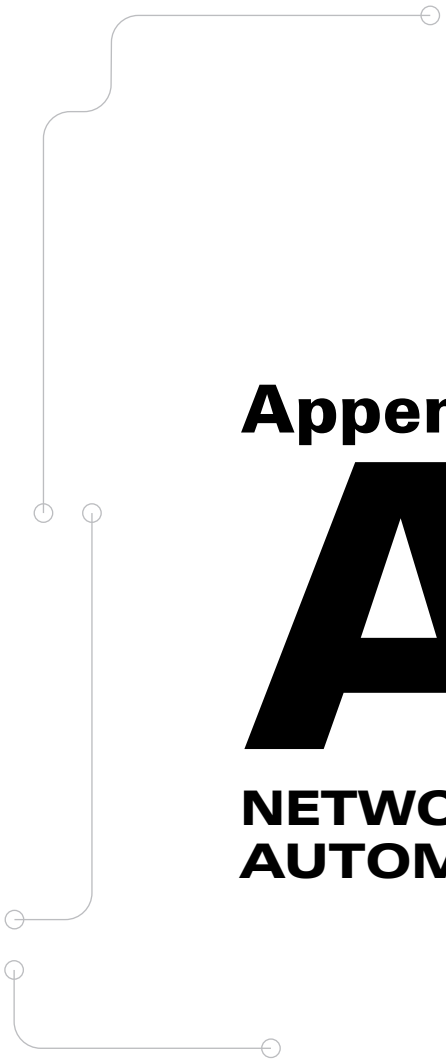
```
+ vault_age: 42
+ vault_weight: "178 pounds"
```

8) Encrypt the vault file using the existing password file:

```
$ ansible-vault encrypt vault_vars.yaml
```

9) Use both files in a short ad hoc play:

```
$ ansible localhost -e @my_vars.yaml -e @vault_vars.yaml -m debug -a 'msg="Name: {{ my_name }}, Age: {{ my_age }}"'
localhost | SUCCESS => {
"msg": "Name: Bob, Age: 42"
}
```



Appendix A

NETWORK AUTOMATION

Network Automation

Not just servers anymore!
Rapidly evolving set of core and vendor provided modules
Key differences

- local execution
- automatic conditional execution of tasks (config diff)
- unique transports
- device specific concept of privilege escalation

Ansible for Network Automation

While Ansible was originally developed with a clear focus on server infrastructure automation, newer releases have an impressive set of functionality specifically targeted at managing network infrastructure. The new features were developed with careful consideration to the workflows and tools commonly used by network admins, and try to build on that foundation so that common tasks can be accomplished within a similar workflow. Per the documentation, the stated goals include:

- Automate repetitive tasks to speed routine network changes and free up your time for more strategic work
- Leverage the same simple, powerful, and agentless automation tool for network tasks that operations and development use
- Separate the data model (in a playbook or role) from the execution layer (via Ansible modules) to manage heterogeneous network devices
- Benefit from community and vendor-generated sample playbooks and roles to help accelerate network automation projects
- Communicate securely with network hardware over SSH or HTTPS

Key Differences

Due to the unique properties of network devices, network device modules have some key differences compared to tradition server management modules. Most of these differences arise due to the devices not supporting Python, which prevents the normal method of Ansible module execution on the managed device. The first attempt to manage devices not supporting Python was the `raw` modules (still an option), but that method is quote limited compared to the new approach. The primary differences are:

Execution on the Control Node ⇒ network module code runs on the control node automatically, as if the `local_action`, or `delegate_to: localhost` control statement was present.

Automatic Conditional Execution ⇒ Network modules do not run every task in a playbook. Instead, the current config state is retrieved and compared to the new config state that would be generated by the described change. If the two configs match, the task is skipped as if a `when: config_has_changed` condition was attached to the task. This attempts to provide idempotency, but has some caveats as described later.

Special Transport Protocols ⇒ In addition to the standard SSH method of connecting to a device, many network vendors offer a REST based API. The network modules offer vendor appropriate transport methods including: XML over SSH, CLI over SSH, API over HTTPS. Some network modules support only one protocol; some offer a choice. The communication protocol used is defined via the `ansible_connection` variable.

Privilege Escalation ⇒ Several network platforms support privilege escalation, often called "enable mode". Ansible 2.5.3 supports become for privilege escalation on eos, ios, and nxos. Needed credentials are defined as host_vars or group_vars files.

The following is an example of the vars needed to fully define escalation parameters. Note that the password can instead be provided via prompt, or encrypted via Ansible Vault:

File: group_vars/cisco

```
ansible_connection: network_cli
ansible_network_os: ios
ansible_become: yes
ansible_become_user: cisco
ansible_become_pass: secret_pass
ansible_become_method: enable
```

Note that versions of Ansible less than 2.5 used a very different method, var dictionary with authorize credentials. If using an older version, consult the docs for details.

Modules

Like all modules, network automation modules have two categories:

1. "core" modules created, and supported by the Ansible project directly:
   http://docs.ansible.com/ansible/latest/modules/network_maintained.html#network-supported
2. "third party" modules (generally created by the network vendors themselves):
   http://docs.ansible.com/ansible/latest/modules/list_of_network_modules.html#network-modules

Simple Network Module Examples

Ad hoc
Playbook equivalent

Ad hoc Network Example

Ad hoc usage of the network modules requires a broader use of Ansible CLI options than the typical interaction with a server. The following example shows using the ios_facts modules to collect info about a compatible Cisco device and explains the options used:

```
$ ansible all -i cisco_device.example.com, -c network_cli ➥
      -u ios_user -K -e ansible_network_os=ios -m ios_facts
```

all ⇒ the host group to which the command should apply
-i name, ⇒ the device to target (trailing comma needed, otherwise interpreted as inventory file)
-c ⇒ connection method
-u ⇒ username (must be defined on end ios device)
-K ⇒ prompt for password
-e ⇒ extra variable to set the network OS
-m ⇒ module name

Many other platform specific variables can be set as described here: http://docs.ansible.com/ansible/latest/network/user_guide/platform_ios.html

Playbook equivalent

File: cisco_facts.yaml

```
---
- hosts: all
  connection: network_cli
  tasks:
  - name: Get facts for IOS devices
    ios_facts:
      gather_subset: all

  - name: Display the facts
    debug:
      msg: "The hostname is {{ ansible_net_hostname }}➥
       and is running image: {{ ansible_net_image }}"
```

```
$ ansible-playbook -i ios.example.com, -u cisco -k ➥
      -e ansible_network_os=ios cisco_facts.yaml
. . . output omitted . . .
```

The following documents the full set of facts returned by this module: http://docs.ansible.com/ansible/latest/modules/ios_facts_module.html#ios-facts-module

Backing up Config

Many network modules offer a backup option that will place a copy of the pre-modified config in the `playbook_root/backup/` directory on the control host. Normal Ansible modules like `copy` (along with `local_action`) can then be used to move the file to a better location if needed. An alternative to get a copy of the config would be the following:

File: `ios_backup_config.yaml`

```
---
- hosts: ios
  gather_facts: no
  vars:
    backup_root: /tmp
  tasks:
  - name: Get running config
    ios_command:
      commands:
      - show run
    register: config

  - name: Get timestamp
    command: date +%Y%m%d
    register: timestamp
    delegate_to: localhost

  - name: Create backup
    copy:
      content: "{{ config.stdout[0] }}"
      dest: "{{ backup_root }}/{{ inventory_hostname }}_
      config_{{ timestamp.stdout }}"
    delegate_to: localhost
```

Network Modules: Gotchas

forks, and strategy
`show running-config` **resource intensive**
SSH jump host has high connection setup overhead
Use canonical command form to avoid `changed=true`

Improving performance: forks and strategy

The default settings of 5 forks, and a parallelism strategy of "linear" can often lead to poor performance when managing network devices. When increasing the forks config property to appropriate scale for a large number of devices managed, remember that network modules run on the control host. This centralized (instead of distributed) load can easily overwhelm the CPU and RAM on the control host if not carefully considered. Also consider switching the strategy from "linear" to "free". The free strategy allows Ansible to use available forks to execute tasks on the devices as quickly as possible instead of requiring each task to complete on the set of hosts before advancing to the next task. Certain operations on network devices can take a long time, and free keeps a single slow task on a host from stalling all progress.

Execute 'show running' only if you absolutely must

The 'show running' command is the most resource-intensive command to execute on a network device, because of the way queries are handled by the network OS. Using the command in your Ansible playbook will slow performance significantly, especially on large devices; repeating it will multiply the performance hit. If you have a playbook that checks the running config, then executes changes, then checks the running config again, you should expect that playbook to be very slow.

Use ProxyCommand only if you absolutely must

Network modules support the use of a proxy or jump host with the ProxyCommand parameter. However, when you use a jump host, Ansible must open a new SSH connection for every task, even if you are using a persistent connection type (network_cli or netconf). To maximize the performance benefits of the persistent connection types introduced in version 2.5, avoid using jump hosts whenever possible.

Abbreviations cause changed=true

When you issue commands directly on a network device, you can use abbreviated commands. For example, `int g1/0/11` and `interface GigabitEthernet1/0/11` do the same thing. However, when abbreviated commands are passed to the device, they will be canonicalized and appear in their unabbreviated form in the config. This means that attempts to diff the config with the proposed set of changes will always result in Ansible thinking a change needs to be made (instead of properly reporting OK).

Simple IOS Modules Examples

```
ios_banner
ios_logging
ios_static_route
ios_{l2_,l3_,}interface
ios_vlan
```

Simple/Targeted IOS Modules

A growing number of targeted modules exist to idempotently configure various specific aspects of the managed node. The following tasks are representative of the usage:

Configure a simple banner:

File: task.yaml

```
- name: Configure banner from file
  ios_banner:
    banner:  motd
    text: "{{ lookup('file', './security_warning.txt') }}"
    state: present
```

Configure the device to send debugging level logs to the syslog server with the specified hostname using local5 facility:

File: task.yaml

```
- name: configure host logging
  ios_logging:
    dest: host
    name: syslog1.example.com
    level: debugging
    facility: local5
    state: present
```

Create a simple static route:

File: task.yaml

```
- name: configure static route
  ios_static_route:
    prefix: 192.168.0.0
    mask: 255.255.0.0
    next_hop: 172.16.2.254
```

Assign an IP address to an interface and then bring it up:

File: task.yaml

```
- name: Set GigabitEthernet0/1 IPv4 address
  ios_l3_interface:
    name: GigabitEthernet0/1
    ipv4: 192.168.0.1/24

- name: make interface up
  ios_interface:
    name: GigabitEthernet0/1
    enabled: True
```

Create a VLAN, assign it to a few interfaces, and configure a bundle of interfaces (defined by a variable) as trunks carrying the specified tagged frames:

File: task.yaml

```
- name: Create vlan
  ios_vlan:
    vlan_id: 100
    name: test-vlan
    state: present

- name: Add interfaces to VLAN
  ios_vlan:
    vlan_id: 100
    interfaces:
      - GigabitEthernet0/0
      - GigabitEthernet0/1

- name: Ensure GigaE0 bundle has vlans 1-10 as trunk vlans
  ios_l2_interface:
    name: "GigabitEthernet0/{{ item }}"
    mode: trunk
    native_vlan: 10
    trunk_vlans: 1-10
  with_items:
  - "{{ trunk_list }}"
```

General Purpose ios Modules

ios_config
ios_command

ios_command

If a network administrator has previously managed devices by interactively running commands on the device, then they can easily leverage knowledge of the ios command syntax and create playbooks that easily run the same commands. Since Ansible has no command exit codes to determine if the command completed successfully, a powerful `wait_for:` construct is provided that allows comparing the automatically captured results and only proceeds when the described tests pass.

Simple command with result comparison:

File: task.yaml

```
- name: Verify that OS verison is IOS
  ios_command:
    commands: show version
    wait_for: result[0] contains IOS
```

Multiple commands checking results from each:

File: task.yaml

```
- name: Verify IOS and Gig 0/{{ g_int }} present
  ios_command:
    commands:
    - show version
    - show interfaces
    wait_for:
    - result[0] contains IOS
    - result[1] contains "GigabitEthernet0/{{ g_int }}"
```

Run command reacting to interactive prompt:

File: task.yaml

```
- name: Reset counters
  ios_command:
    commands:
    - command: 'clear counters GigabitEthernet0/2'
      prompt: 'Clear "show interface" counters ↵
      on this interface [confirm]'
      answer: c
```

Wait for up to 30 seconds for a specific OSPF route to be seen in the local table:

File: task.yaml

```
- name: Wait for route to be seen
  ios_command:
    commands:
    - show ip route
    interval: 3
    retries: 10
    wait_for:
    - result[0] contains "O 172.16.1.0 [110/11] via 1.1.1.2"
```

Direct Config Manipulation with ios_config

Instead of passing a list of specific commands to execute, config file manipulations can be described. Given the fact that the config is for the most part a list of commands that could be entered in that order to produce the current config state, the effects of the ios_command and ios_config modules are obviously intimately related.

The ios_config module is more complicated to use. It has a variety of functions to backup and diff configs, but the parameters you will want to focus on are:

before ⇒ commands to execute first if a change needs to be made
lines ⇒ lines in the configuration for the described section
after ⇒ commands to execute after lines, if a change is to be made
match ⇒ how to match or compare the "lines" and corresponding config section
parents ⇒ parents that uniquely identify the section to be checked/modified
replace ⇒ how to perform replace operations (individual lines, or entire block)

Idempotency and Canonicalization of Input

The simplest task is perhaps adding a single line; for example:

File: task.yaml

```
- name: Permit SSH
  ios_config:
    lines: access-list 180 permit tcp any any eq 22
```

However, the above task will result in a change every time it is run (not idempotent). This is because ios accepts the port as a number, but displays it as a service name. Instead, you would need to use the following:

File: task.yaml

```
- name: Permit SSH
  ios_config:
    lines: access-list 180 permit tcp any any eq ssh
```

Using the before Parameter

To run commands before a change, but only if a change is needed, use the before parameter. For example, if an ACL does not match exactly the specified lines, remove it and then add the desired config:

File: task.yaml

```
- name: Configure ACL
   ios_config:
     parents: "ip access-list extended demo-acl"
     lines:
     - permit tcp any any eq www
     - permit udp any any eq snmp
     before: "no ip access-list extended demo-acl"
```

Using the after Parameter

Some commands change the configuration of the device, but are not reflected in the normal show run output. Including these as normal lines will again result in a change being triggered every run. The after parameter can be used to run the needed command only when the designated lines cause a change for example:

File: task.yaml

```
- name: Configure SNMPv3
  ios_config:
    lines:
    - snmp-server group SNMPv3 v3 priv
    after:
    - snmp-server user myuser SNMPv3 v3 auth md5 myPass
```

Using the parents Parameter

File: task.yaml

```
- name: Configure Gig1/1
  ios_config:
    parents: "interface GigabitEthernet1/1"
    lines:
      - description To Core
      - ip address 172.16.0.1 255.255.255.0
```

Using the replace Parameter

The line mode for replace can result in surprising results when used in combination with the before parameter. For example, consider a config that already contains the following ACL:

File: running_config

```
ip access-list extended input
 permit tcp any any eq www
```

If the following task is applied, the result will NOT be both ACL lines, but rather only the one pertaining to SMTP. The reason is that the diff operation is first performed, with results identifying the WWW line already being present. The change is triggered, but the before parameter clears the entire list, and then only the SMTP line is added (since line mode is specified). Instead, the task should be modified so that "replace: block" is used, which will add all config lines listed:

File: task.yaml

```
- name: Configure TEST-ACL
  ios_config:
    parents: "ip access-list extended TEST-ACL"
    lines:
    - permit tcp any any eq smtp
    - permit tcp any any eq www
    before: "no ip access-list extended TEST-ACL"
    match: exact
    replace: line
```