### What is Ansible?

Ansible is a free software for configuring and managing nodes. It easy to use and includes many built-in modules to allow easy configuration management. It uses ssh to connect the different nodes and configure them. The only thing you need on your machine in order to run Ansible is python installed ( > 2.4).

**Task**

A task is simply the use of one of Ansible modules. Module implements specific functionality.  For example, installing a package would be a task since it will require us to use the ‘yum’ module. There are many modules, so a task can be running a service, fetching files, adding user and many more modules waiting for you to explore.

Let’s see how task looks like:

- name: Ensure python-ryu is installed

yum:

name: python-ryu

state: present

In the above example we use yum module to install a package named ‘python-ryu’. The state is the action we are using on this package. so ‘present’ tells Ansible to make sure python-ryu installed in the system. There are additional states, as ‘latest’ which means ‘make sure latest package is installed’, so if you have python-ryu-1.0 installed in your system, but there is python-ryu-2.0 available, it will be updated. This is not the case for ‘state: present’, which simply cares on whether the package is installed or not.

**TIP:**to see what states and other options available for yum module, use this command: ansible-doc yum

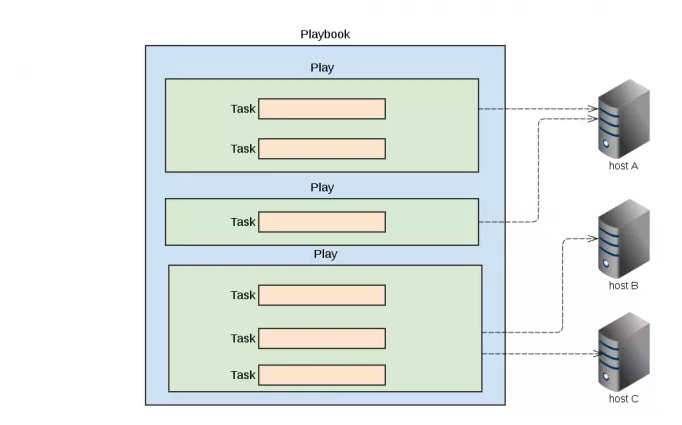
**Play**

Play is a collection of tasks running on one or more hosts. It includes one or more task.

**Playbook**

Playbook composed of one or more plays.

It’s important to be familiar with the relation between task, play and and playbook:



### Step 1: Installing Ansible

You can install Ansible with:

apt-get install ansible

You need to put all the servers that you want to manage with Ansible in the /etc/ansible/hosts file.

You will need to comment out all lines. Go to the latest line of the hosts file to create a category. Say you have a cluster of web and database servers. You could create two separate categories: web and db. If you would want to make a change on all database servers, you could use db as selection so only all database servers would be affected and not other servers such as your web servers in the web category.

Example:

[web]

localhost ansible\_ssh\_host=127.0.0.1

web1 ansible\_ssh\_host=192.168.2.2

web2 ansible\_ssh\_host=0.0.0.0

[db]

db1 ansible\_ssh\_host=192.168.2.3

db2 ansible\_ssh\_host=192.168.2.4

db3 ansible\_ssh\_host=192.168.2.5

db4 ansible\_ssh\_host=192.168.2.6

Format: name ansible\_ssh\_host=ip

Note that if you're using Ansible 2.0, the ssh\_ part has been deprecated. Instead use ansible\_host.

name is just a name to refer to your server, ip is the actual IP.

This tells Ansible that you have 3 web servers on the IP addresses 127.0.0.1, 192.168.2.2 and 0.0.0.0 and 4 database servers on the IP addresses 192.168.2.3-6.

### Step 2: Setting up SSH keys

**SETTING UP PUBLIC KEY AUTHENTICATION**

Key based authentication in SSH is called public key authentication. The purpose of ssh-copy-id is to make setting up public key authentication easier. The process is as follows.

**GENERATE AN SSH KEY**

With OpenSSH, an SSH key is created using ssh-keygen. In the simplest form, just run ssh-keygen and answer the questions. The following example illustates this.

### COPY THE KEY TO A SERVER

Once an SSH key has been created, the ssh-copy-id command can be used to install it as an authorized key on the server. Once the key has been authorized for SSH, it grants access to the server without a password.

Use a command like the following to copy SSH key:

ssh-copy-id -i ~/.ssh/mykey user@host

This logs into the server host, and copies keys to the server, and configures them to grant access by adding them to the authorized\_keys file. The copying may ask for a password or other authentication for the server.

Only the public key is copied to the server. The private key should never be copied to another machine.

### TEST THE NEW KEY

Once the key has been copied, it is best to test it:

ssh -i ~/.ssh/mykey user@host

The login should now complete without asking for a password. Note, however, that the command might ask for the passphrase you specified for the key.

### TROUBLESHOOTING

There are a number of reasons why the test might fail:

* The server might not be configured to accept public key authentication. Make sure /etc/ssh/sshd\_config on the server contains PubkeyAuthentication yes. Remember to restart the sshd process on the server.
* If trying to login as root, the server might not be configured to allow root logins. Make sure /etc/sshd\_config includes PermitRootLogin yes, PermitRootLogin prohibit-password, or without-password. If it is set to forced-commands-only, the key must be manually configured to use a forced command (see command= option in ~/.ssh/authorized\_keys.
* Make sure the client allows public key authentication. Check that /etc/ssh/configincludes PubkeyAuthentication yes.
* Try adding -v option to the ssh command used for the test. Read the output to see what it says about whether the key is tried and what authentication methods the server is willing to accept.
* OpenSSH only allows a maximum of five keys to be tried authomatically. If you have more keys, you must specify which key to use using the -i option to ssh.

## HOW SSH-COPY-ID WORKS

ssh-copy-id uses the SSH protocol to connect to the target host and upload the SSH user key. The command edits the authorized\_keys file on the server. It creates the .ssh directory if it doesn't exist. It creates the authorized keys file if it doesn't exist. Effectively, ssh key copied to server.

It also checks if the key already exists on the server. Unless the -f option is given, each key is only added to the authorized keys file once.

It further ensures that the key files have appropriate permissions. Generally, the user's home directory or any file or directory containing keys files should not be writable by anyone else. Otherwise someone else could add new authorized keys for the user and gain access. Private key files should not be readable by anyone else.

## SOME BEST PRACTICES FOR SSH KEYS

SSH keys are very useful, but can lead to problems if they are not properly managed. They are access credentials just like user names and passwords. If they are not properly removed when people leave or systems are decommissioned, no-one may any longer know who really has access to which systems and data. Many large organizations have ended up having millions of SSH keys.

### USE A PASSPHRASE WHEN POSSIBLE

It is recommended that keys used for single sign-on have a passphrase to prevent use of the key if it is stolen or inadvertatly leaked. The ssh-agent and ssh-add programs can be used to avoid having to enter the passphrase every time the key is used.

Generally all keys used for interactive access should have a passphrase. Keys without a passphrase are useful for fully automated processes. They allow shell scripts, programs, and management tools to log into servers unattended. This is often used for backups and data transfers between information systems.

### ADD A COMMAND RESTRICTION WHEN POSSIBLE

The copy-id tool does not automatically add command restrictions to keys. Using command restrictions is highly recommended when the key is used for automating operations, such as running a report for fetching some files. A command restriction is basically a command="<permitted command>" option added to the beginning of the line in the server's authorized\_keys file.

### MANAGING SSH KEYS

Anyone having more than a few dozen servers is strongly recommended to manage SSH keys. Not managing the keys exposes the organization to substantial risks, including loss of confidentiality, insertion of fraudulent transactions, and outright destruction of systems.

The copy-id tool can be dangerous. It can easily accidentally install multiple keys or unintended keys as authorized. The logic for choosing which key to install is convoluted. Extra authorized keys grant permanent access. They can later be used to spread attacks host-to-host, and the more keys there are, the higher the risk. It also violates all regulatory compliance requirements

### Step 3: Test Ansible

To see if you can ping all your servers in the hosts file, you can use the following command:

ansible -m ping all

This confirms whether or not your servers are online.

You can also execute a command:

ansible web -m command -a 'shutdown -h now'

We've just executed the command shutdown -h now on all servers in the web category.

### What is a Playbook?

Playbooks are essentially sets of instructions (plays) that you send to run on a single target or groups of targets (hosts). Think about the instructions you get for assembling an appliance or furniture. The manufacturer includes instructions so you can put the parts together in the correct order. When followed in order, the furniture looks like what was purchased.   
  
That's basically how a playbook works.

**Modules**

The Playbook we're building will install a web server on a target RHEL/CentOS 7 host, then write an index.html file based on a template file that will reside with the final Playbook. You'll be able to take the example Playbook and additional files from this blog and test it out for yourself. While going over the example Playbook, we'll explain the modules that are used.

**Authors**

The author adds instructions for the modules to run, often with additional values (arguments, locations, etc.). The target host has modules run against it in the order the Playbook lays out (with includes or other additional files). The host's state is changed (or not) based on the results of the module running, which Ansible and Tower displays in output.

### Running Playbooks

Keeping that in mind, you're still going to need to understand a few things about running Playbooks. With the furniture analogy, a Playbook is shorthand to tell the modules to perform a task. You must understand the following to run your Playbook successfully:

**1. The target**

Because the Playbooks are providing direction and interactivity with the modules, Ansible assumes you know how to do what you're trying to do and automates it. That's why Playbooks are like instructions or directions - you're telling the automated parts how you want the task configured. You'll still need to understand the target you're running the Playbook against.

**2. The tasks**

If part of the Playbook needs to start the web server, you're going to need to know how that's done so you know to use the service module and start the web server by name. If the Playbook is installing software, then you have to know how installation is done on the target. You're also still going to need to understand the basics of the tasks being performed. Does the software you're installing have a configuration setup to it? Are there multiple steps that require conditions or argument values? If there are variables that are being passed, these will all need to be understood by those constructing a Playbook as well.

### ****Writing your first playbook****

Before writing any lines, we need to have clear picture of what we want to run on our hosts and how we want  them to look after running the playbook.

So imagine our environment consists of two hosts:  hostA and hostB

We want to have two plays:

First play: one simple task – create file in /tmp named ‘yallo’.

Second play: two tasks  – task to add user named ‘mario’ with zsh as default shell, and anther task to install the latest  ‘zlib’ package.

We know what we want to run and on what hosts we want to run it. All that is left is to write it. Let’s start with the first play:

vi first\_playbook.yml

---

- hosts: hostB

tasks:

- name: Create file

file:

path: /tmp/yallo

state: touch

- hosts: all

tasks:

- name: Ansible create file if it doesn't exist example

file:

path: "/Users/mdtutorials2/Documents/Ansible/devops\_server.txt"

state: touch

Let’s now extend it to include the second one also:

---

- hosts: hostB

tasks:

- name: Create file

file:

path: /tmp/yallo

state: touch

- hosts: my\_hosts

sudo: yes

tasks:

- name: Create user

user:

name: mario

shell: /bin/zsh

- name: Install zlib

yum:

name: zlib

state: latest

---  
- hosts: all  
  sudo: true  
  vars:  
     packages: [ 'vim', 'git', 'curl' ]  
  tasks:  
     - name: Install Package  
       apt: name={{ item }} state=latest  
       with\_items: packages

---  
- hosts: all  
  sudo: true  
  vars:  
     packages: [ 'vim', 'git', 'curl' ]  
  tasks:  
  - name: Install Package  
    apt: name={{ item }} state=latest  
    with\_items: packages  
---  
- hosts: all  
  tasks:  
     - name: Install httpd Package  
       apt: name=httpd state=latest

---

- name: Install nginx

hosts: host.name.ip

become: **true**

tasks:

- name: Add epel-release repo

yum:

name: epel-release

state: present

- name: Install nginx

yum:

name: nginx

state: present

- name: Insert Index Page

template:

src: index.html

dest: /usr/share/nginx/html/index.html

- name: Start NGiNX

service:

name: nginx

state: started

There is a good chapter in the docs: Variable File Separation.

Define users.yml:

users:

- name: testuser3

comment: testuser3

groups: wheel

- name: testuser4

comment: testuser4

groups: users

And in your playbook:

- hosts: servers

remote\_user: root

vars\_files:

- /vars/users.yml

tasks:

- name: adding several users

user:

name: "{{ item.name }}"

state: present

comment: "{{ item.comment }}"

password: "$1$\*)^%$CeUUJM&v#0pQhHjqjpqX1"

update\_password: on\_create

createhome: yes

groups: "{{ item.groups }}"

with\_items: "{{ users }}"

### Playbooks

ansible-playbook <YAML> # Run on all hosts defined

ansible-playbook <YAML> -f 10 # Run 10 hosts parallel

ansible-playbook <YAML> --verbose # Verbose on successful tasks

ansible-playbook <YAML> -C # Test run

ansible-playbook <YAML> -C -D # Dry run

ansible-playbook <YAML> -l <host> # Run on single host

Run Infos

ansible-playbook <YAML> --list-hosts

ansible-playbook <YAML> --list-tasks

Syntax Check

ansible-playbook --syntax-check <YAML>

### Remote Execution

ansible all -m ping

Execute arbitrary commands

ansible <hostgroup> -a <command>

ansible all -a "ifconfig -a"

### Debugging

List facts and state of a host

ansible <host> -m setup # All facts for one host

ansible <host> -m setup -a 'filter=ansible\_eth\*' # Only ansible fact for one host

ansible all -m setup -a 'filter=facter\_\*' # Only facter facts but for all hosts

Save facts to per-host files in /tmp/facts

ansible all -m setup --tree /tmp/facts