- Ansible Components
  - Inventory
  - Playbook
  - Modules
  - Roles
- Ansible Galaxy

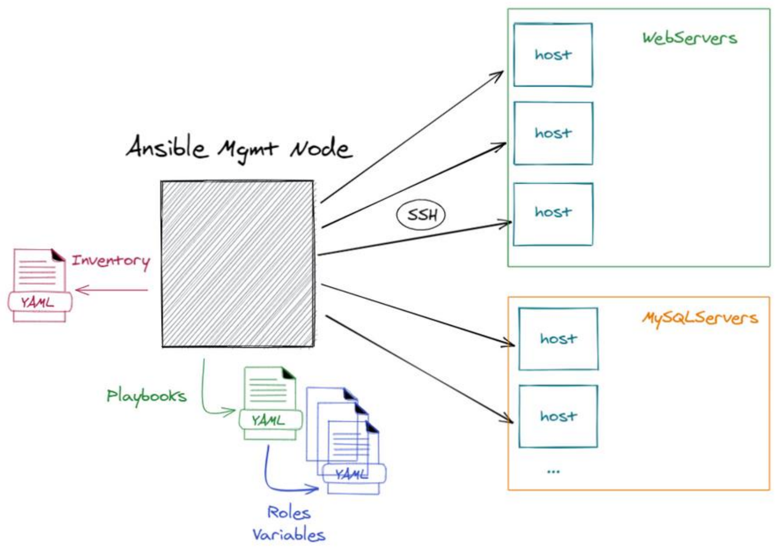
What is Ansible?

Ansible --> open-source configuration management and provisioning tool similar to puppet and chef.
It uses SSH to connect to servers and run configured tasks, without the need to install any agent on the other servers.

Why Ansible?

No Agent to install, just SSH.
Idempotent, not just about running a whole script in sequence, but it is about a system state to achieve, it checks what to change, what to apply, if the configuration is already satisfied then nothing will be changed.
also desired state configuration is about how do you like things to be done? And the answer will always be in past tense as I want a service to be started or stopped, a software to be installed or removed.
Declarative تعتمد على التوصيف, not procedural (follow sequence), Ansible focus on describing the state of the machine, then it applies the steps that fulfill that description.
Easy to learn, doesn't take time to use it.

How Ansible Works?



All you need an SSH between master node and the hosts.

- **ssh -i key-file user@ip**

or set password less authentication between the master and hosts.

On the master do:

- **ssh-keygen**
- **ssh-copy-id -i /root/.ssh/id_rsa.pub user@ip**

if already obtained a key from cloud or anywhere else then it is important to do this

- **chmod 400 key-file**

notice that ssh configuration is per user, can be found at the home directory under ~/.ssh

Ansible Installation

First thing to do is update the package manager metadata for example in ubuntu: **apt-get update**
After that we can use **apt-get install ansible**
Ansible is a python package, so if we have python installed along with pip we can do:

- **pip install ansible**

to check that ansible is installed we use **ansible –version**

INVENTORY

Common formats include **INI** and **YAML**
It is a description of the nodes (IP addresses / host names/FQDN) that can be accessed by ansible, by default its configuration file is under **/etc/ansible/hosts ,** so if there is not any inventory given in the command, ansible will read from this file.

Nodes or hosts can be assigned to groups. As **[webservers] , [db-servers].**
We can also make nested groups by using the suffix :children with groups name for example:

[project1:children]

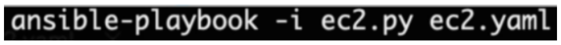
Webservers
db-servers

EXAMPLE: ANSIBLE INVENTORY FILE WITH AWS EC2 instance record

Alias **ansible_host=**<public DNS record> **ansible_port=**<port> **ansible_user=**<ec2-user>
ansible_ssh_private_key_file=<path to private key file>

for security reason the private key file should have only read permission for the file owner
chmod 400 key-file

inventory files can be dynamic through scripts as:



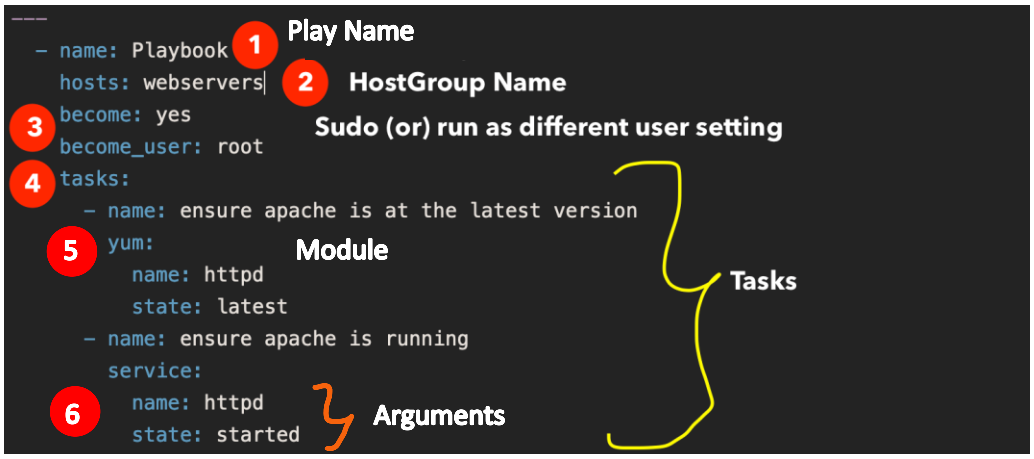
```
ansible-playbook -i ec2.py ec2.yaml
```

Why not to use the default inventory?
Because we need configuration to be dynamic and customized per project, it is recommended as well to have separate inventory files for different environments as dev, prod ,etc.
To test that we set the inventory file correctly we can do the following:
ansible <targeted host-name in the inventory> -i <inventory file> -m <module name to run on the host>

PLAYBOOK

Just a YAML file, to describe the desired state of the system.

Playbook contains **plays**, plays contain **tasks**, tasks call **modules**.



```
---
- name: Playbook
  hosts: webservers
  become: yes
  become_user: root
  tasks:
    - name: ensure apache is at the latest version
      yum:
        name: httpd
        state: latest
    - name: ensure apache is running
      service:
        name: httpd
        state: started
```

Ansible has more than 1000 module, to automate actual wok, they are what gets executed in each task. Example of modules: **service, copy, iptables, yum, command, file** etc.

AD-HOC COMMAND

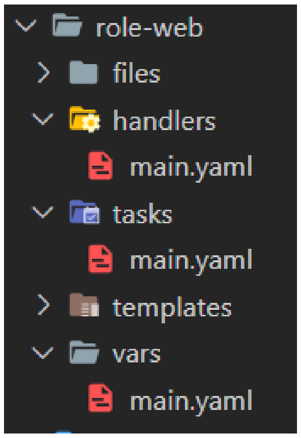
It is a quick way of testing modules, without the need of inventory file or play book

- ansible localhost -m module-name -a arguments(space separated)

ROLES

It is a way to group tasks into one container, to make tasks more organized and reusable across multiple projects, Roles can be published on centralized place to other people called **Ansible Galaxy**

Role is all about a folder with a specific structure:



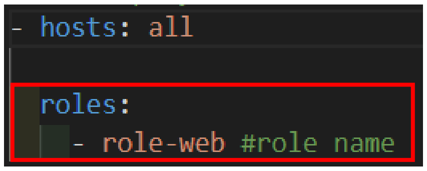
```
role-web
  files
  handlers
    main.yaml
  tasks
    main.yaml
  templates
  vars
    main.yaml
```

Role folder:

- files --> files that will be used with copy module.
- handlers
- tasks
- templates --> files that will be used with template module.
- vars

Roles purpose is more organized structure that’s all.

To use the role in the playbook all we do is :



```
- hosts: all

  roles:
    - role-web #role name
```

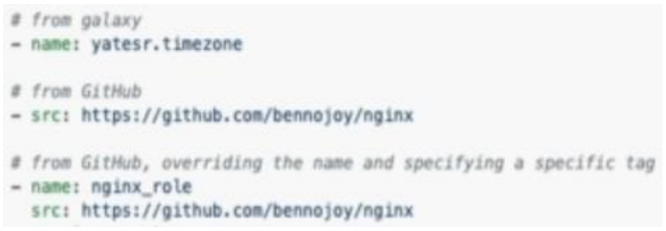
Ansible Galaxy is a place where other people have published their roles as well to be used and installed.

To install a role from there: **ansible-galaxy install {role-name}**

To create the previous structure with the help of Ansible galaxy: **ansible-galaxy init {role-name}**

Download Multiple Roles

In real work environment Roles can be in Ansible Galaxy, GitHub, Gitlab or bitbucket or any source control, we have already known imperatively how to install roles from CLI, now it is the time for the declaratively way (which is the recommended way by ansible to do), through a file in the role folder called **requirements.yaml** like this Example:



```
# from galaxy
- name: yatesr.timezone

# from GitHub
- src: https://github.com/bennojoy/nginx

# from GitHub, overriding the name and specifying a specific tag
- name: nginx_role
  src: https://github.com/bennojoy/nginx
```

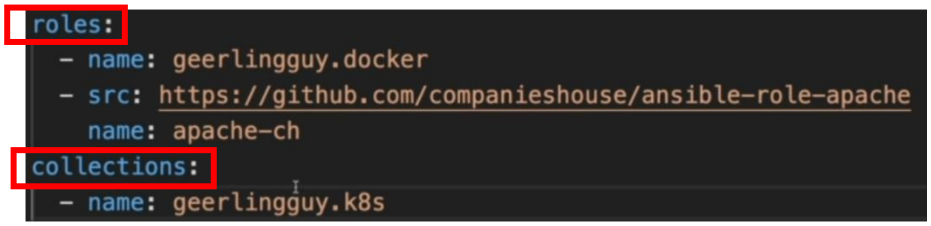
Now we have prepared everything, it is time for installing those roles, we can do that through this command:

ansible-galaxy install --roles-path <path-where-to-install> -r requirements.yaml

so far, we have known what roles are, which is an individual thing on its own, now we will discover collections, a collection consists of multiple roles.

To install a collection imperatively, we use **ansible-galaxy collection install <collection-name>**

We can do the installation declaratively as well with the help of **requirements.yaml** and here is an example:



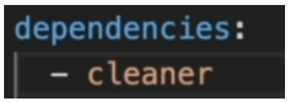
```
roles:
  - name: geerlingguy.docker
  - src: https://github.com/companieshouse/ansible-role-apache
    name: apache-ch
collections:
  - name: geerlingguy.k8s
```

For installation we use **ansible-galaxy collection install -p <path-where-to-install> -r requirements.yaml**

In most cases **Roles** are always the best choice to use over **Collections.**

Dependencies

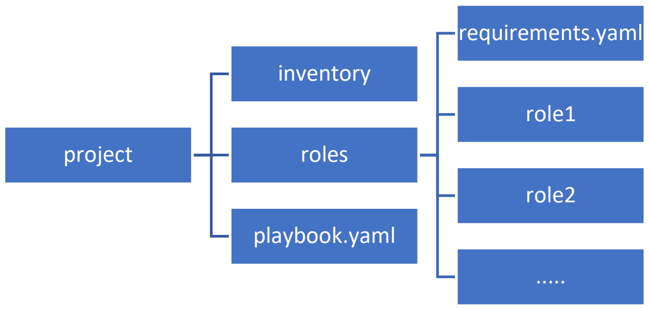
Inside **meta** folder in **role** we can specify dependencies for our role, like we might need a certain role to end its job completely before another role starts.



```
dependencies:
  - cleaner
```

So having dependencies like this example in Apache role we are saying that for Apache role to start their must be a cleaner role to run before.

Ansible project Final Structure



How to run a playbook?

- **ansible-playbook -i** {inventory_file} **--private-key** {key_file} {playbook_file}

If we already provided private key in the inventory file, so no need to provide it in the above command, also we can add **--ask-become-pass** or **--ask-vault-pass** in the CLI to prompt for the remote user password.

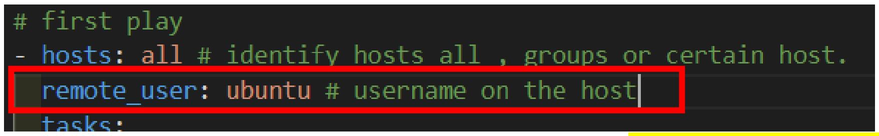
First stage in running a playbook is **gathering facts,** which test the connectivity between host and ansible server, and save information about the remote servers as an object --> **ansible_facts**

If host's user wasn't specified in inventory file or playbook, then there would be an assumption that the user would be as the ansible server, and that might raise an error.

To solve it either in **inventory file**:

```
Webserver1 ansible_host=192.168.66.189 ansible_user=cloud
```

Or **playbook file:**



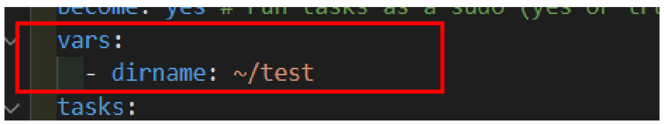
```
# first play
- hosts: all # identify hosts all , groups or certain host.
  remote_user: ubuntu # username on the host
  tasks:
```

remote_user represents the user that the ansible master will use to establish a connection with the remote hosts remote_user@host, refers to the user account that exists on the target host

- **Variables**
- **Debug**
- **Template**
- **Conditions**
- **Loops**
- **Handlers**

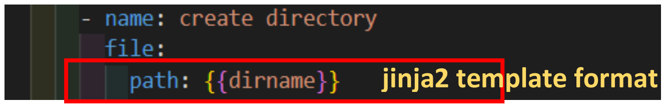
VARIABLES

Declare variables and assign values:



```
become: yes # run tasks as a sudo (yes or tr
vars:
  - dirname: ~/test
tasks:
```

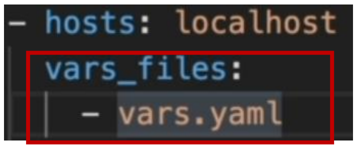
Use variables:



```
- name: create directory
  file:
    path: {{dirname}}
```

For critical data, use **vars_prompt** instead of vars, for data like passwords to be prompted in the run time to be given.

Variables can be created in separated file that contains only the variables, and will be populated on the playbook run, variables file can be given



```
- hosts: localhost
  vars_files:
    - vars.yaml
```

- manually when providing the file **-e "@var-file".**
- declarativity inside the playbook using **vars_files.**

DEBUG

كنا قولنا ان اللى بيميز ال Ansible انها Idempotent بمعني انها بتهتم انها توصل ل state معينه على انها ت run ال script وخلاص من غير اى check على حاله ال system الفعليه.

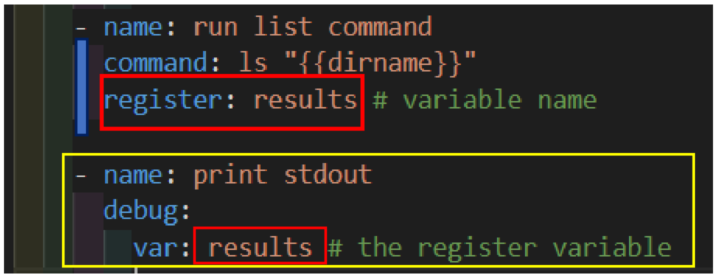
بس مش كل ال modules فى Ansible بتطبق ال Idempotency بمعني ان هدفها ت run وخلاص زى ال **command module .**

فى حاله انى استخدمت module زى ده, فى الغالب هكون عاوز اشوف ال output بتاعه ولو جربنا باللى احنا نعرفه, هنلاقي ان مطلعليش output من ال module يعرفني ايه اللى حصل.

لو انا فى module مهتم اشوف ال standard output بتاعه ساعتها هستخدم ال **register & debug .**

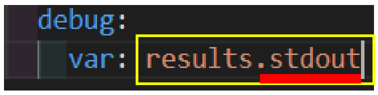
Register: parameter but not associated to a specific module, written on the indentation level of the task.

Debug: module in ansible, to get benefits of register variables.



```
- name: run list command
  command: ls "{{dirname}}"
  register: results # variable name

- name: print stdout
  debug:
    var: results # the register variable
```

Debug returns detailed JSON format, if we need just the output then the debug should be like that:



```
debug:
  var: results.stdout
```

Or stdout_lines

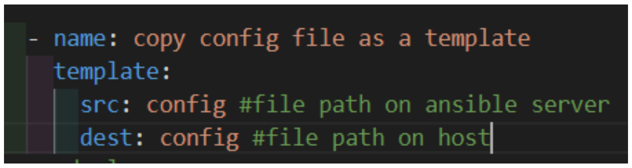
TEMPLATE

Standard way of not repeating configuration across multiple hosts, that share a same configuration file, with different values, we can get benefits of templates through variables, conditions, and loops.

We have two modules for copying:

- **Copy** --> copy the content of src as it is.
- **Template** --> copy the content as well but check for variables in the process, if any was found, will be mapped to the given value in the play book **vars** section.

Config here was a file, that contains few lines, some were variables, and their values were provided in the vars section.



```
- name: copy config file as a template
  template:
    src: config #file path on ansible server
    dest: config #file path on host
```

Template config file extension is: **.j2**

Another important argument in copy or template is **remote_src** which is a Boolean so

- If `false`, it will search for `src` on the contr   oller node.
- If `true` it will search for `src` on the managed (remote) node itself.

In all situation dest is the managed remote host.

CONDITIONS

Logical flow control, known as "if condition" at any programming language, but in ansible, it is called "**when**", written in the same task level indentation.

Note: conditions has no relation with the module, so it wouldn't be found with the module arguments.

Conditions can be used on custom variables or facts. If condition is not matched is the task would be skipped.

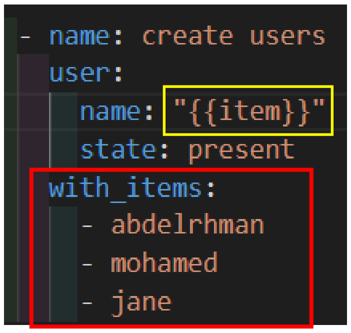
We can use logical operators with conditions as and, or.

LOOPS

Used to shorten your syntax, by using a certain code over a different value each time.

This could be achieved through something like this example:

Same task, different values become easier now.



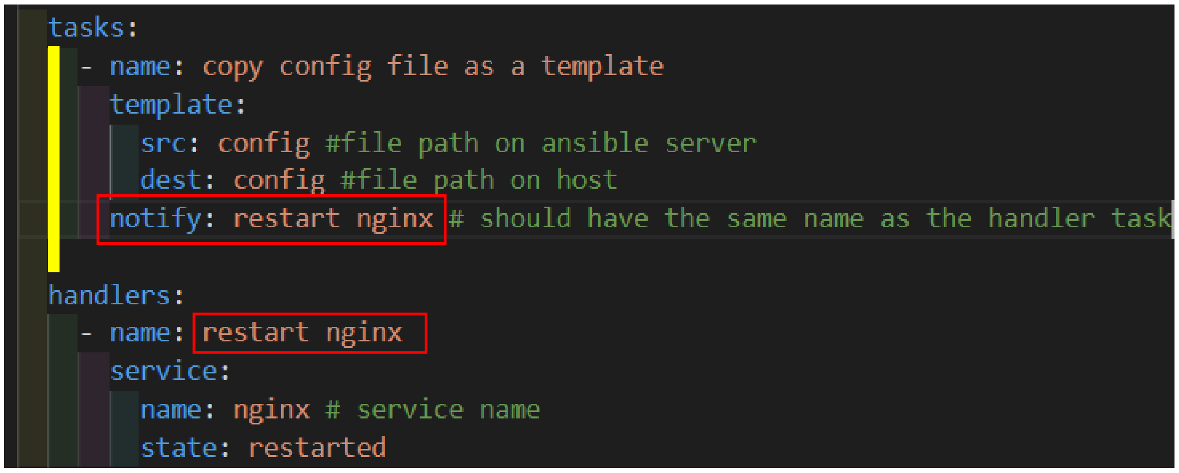
```yaml
- name: create users
  user:
    name: "{{item}}"
    state: present
  with_items:
    - abdelrhman
    - mohamed
    - jane
```

Another module that does the same as with_items is loop.

Loop module is the newer version, just replace with_items with loop.

HANDLERS

Used to run tasks as well, but tasks that are dependent on other tasks to finish their job first or tasks th at made any change, and it runs at the end of the playbook execution.



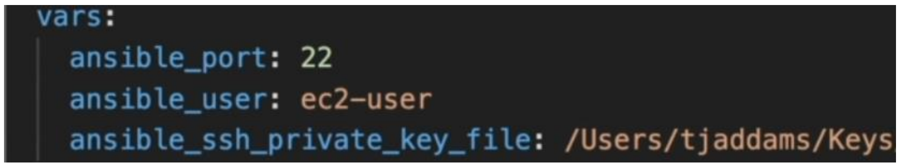
```yaml
tasks:
  - name: copy config file as a template
    template:
      src: config #file path on ansible server
      dest: config #file path on host
    notify: restart nginx # should have the same name as the handler task

handlers:
  - name: restart nginx
    service:
      name: nginx # service name
      state: restarted
```

Handlers contains tasks that would not run frequently, tasks that run only when a change occurs, so in the above example, if there is any change happens in the config file the state would be changed, meanwhile the task that caused the change will **notify** the handler task to do it is job.

Handlers run only one time at the end of playbook execution, no matter how tasks notify the handler.

EXTRA knowledge

- If we are going to use dynamic inventory file, then it would be good to pass some variables as we were doing in static files.

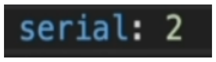


```
vars:
  ansible_port: 22
  ansible_user: ec2-user
  ansible_ssh_private_key_file: /Users/tjaddams/Keys
```

- Now we know that ansible can run on large number of hosts, we might want to go incrementally working with huge number of hosts like we configure 10 hosts each time or increase to 20 and so on till the end, we can achieve that using **serial** in the playbook.

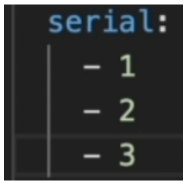
Here are some examples to configure number of hosts:

- at a time



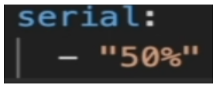
```
serial: 2
```

- scaling incrementally



```
serial:
  - 1
  - 2
  - 3
```

- perectenage



```
serial:
  - "50%"
```

- We might have noticed that in our ansible files, there can be sensitive data like passwords, there is a solution offered but not the best, which **ansible-vault,** all it does that it encrypts the content of a given file.

Example **ansible-vault encrypt <**file name**>**

And to view the content we run **ansible-vault view <**file name**>** but the file itself will still be encrypted, If we actually want to decrypt the file we run **ansible-vault decrypt <**file name**>**

We can change the vault password as well through **ansible-vault rekey <**file name**>**

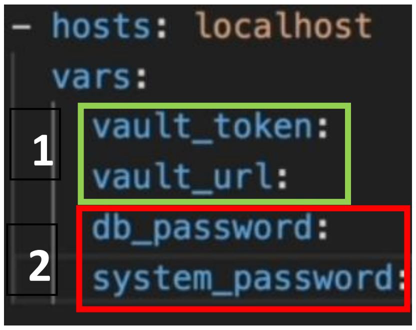
while running the playbook we will need to provide the vault password through **--ask-vault-ask**

- now with the alternative of ansible-vault which is **hashi_vault module**, we just need to install **hashicorp vault** it enables more functionality and security measures than what we had before, it is main idea to focus on the secrets we want rather than encrypting the whole file.

The only requirement for using hashi_vault is a python library called **hvac.**

To start Vault after we have installed it, we run vault server -dev it is insecure to run dev mode in production environment

Example working with vault



```
- hosts: localhost
  vars:
    vault_token:
    vault_url:
    db_password:
    system_password:
```

vault in playbook

after installing **hashicorp vault** , now we can use hashi_vault module which is a lookup module that query the vault API to return a value.

To start a vault run `vault server -dev` , vault works with key value store (key = value) which will be our secrets.

Secrets are stored in paths, example:

- Secrets/passwords/db
- Secrets/passwords/system

To work with vault, we need to provide two important pieces of information:

- The vault token
- The vault URL

Types of keys provided in vault

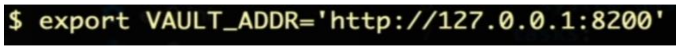
- Unseal key needs to be provided when we start a vault in normal mode

In dev mode the vault become automatically unsealed

- Root Token is the most important and need to be safe, we won't be using it in real production every project/app will its own specific key

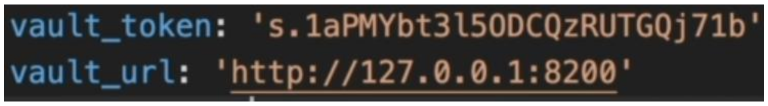
To use vault in our playbook we need to provide a the vault token and vault URL, we get these information while starting the vault as we discussed so in our example:

- The vault_token will be the root token
- The vault_url will be the VAULT_ADDR which will be an environment variable that we set in this way:



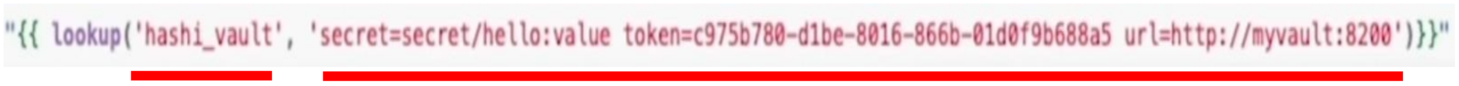
```
$ export VAULT_ADDR='http://127.0.0.1:8200'
```

doing that the first part is completed



```
vault_token: 's.1aPMYbt3l5ODCQzRUTGQj71b'
vault_url: 'http://127.0.0.1:8200'
```

moving to the second part which is dealing with secrets and how to retrieve them from vault, this is an example of what we need to tell ansible how to retrieve a secret:



```
"{{ lookup('hashi_vault', 'secret=secret/hello:value token=c975b780-d1be-8016-866b-01d0f9b688a5 url=http://myvault:8200')}}"
```

in our example we have:

```
db_password="{{ lookup('hashi_vault' , 'secret=<secret path:key> token={{vault_token}} url={{vault_url}}')}}"
```

in this we are telling ansible that we use a lookup module, and it is called **hashi_vault ,** the lookup( ) takes two parameters the lookup module to be used and some attributes as **secret**, **token** and **url.**

Now we just missing to set our secrets, to do so we have two ways either from

- Vault UI using the vault url
- CLI using the command **vault kv put <**secret path**>** key**=**value

If we want to use the CLI we might need to change the KV engine version from V1 to V2 (V2 allows versioning, that we have different version from a secret) through these commands:

1. **vault secrets disable secret**
2. **vault secrets enable -version=1 -path=secret kv**