Introduction to Ansible

In short, Ansible is a versatile tool which can do configuration manamgement, provisioning, remote execution and a lot of other jobs.

What is Ansible?

Ansible is a modern automation tool which makes your life easier by managing your servers for you. You just need to define the configuration in which you are interested and ansible will go ahead and do it for you, be it installing a package or configuring a server application or even restarting a service. Ansible is always ready to manage your servers.

Why do we need it?

Managing a server is easy. Managing 5 is do'able. Managing hundreds or more is a painful task without automation. Ansible is designed to be simple and effective. You can create identical, replicable servers and clusters of servers in painless and reliable manner.

What are the advantages of using it?

- Ansible is agentless. You do not need to have anything installed on the client's end.
- Both push and pull mode are supported.
- It uses OpenSSH as transport protocol.
- Ansible scripts (commonly known as playbooks) are writting in yml and are easy to read.
- Hundreds of modules and thausands of roles freely available to use.
- Supports bare-metal and cloud out of the box.

How to install Ansible?

We will install the Ansible by yum. Package managers like dnf, yum and apt can be used to obtain a stable version and usually pip provides latest version.

```
# yum -y install epel-release

# yum -y install ansible git vim
```

Inventory File

Inventory defines the list of the hosts that we are going to manage. We can group them logically in certain roles. For example, we would want to group our web servers in one group and application servers in another. A group can have multiple servers and one server can be a part of multiple groups.

It is also possible to have a group of groups by using :children suffix. Variables like the user which will be sued to execute the playbook commands can be defined for hosts as well.

Name of group is enclosed in square brackets []. Server names can be their dns names or ip addresses.

```
[webservers]

server1

[application]

server1

server2    ansible_user=aditya

[cluster:children]

webservers

application
```

The path to default inventory file is defined in Ansible configuration `/etc/ansible/ansible.cfg` but it is possible to use any file as inventory by passing flag `-i` to the ansible command. For dynamic environments like EC2, there are plugins and modules to generate inventory dynamically.

Ansible Modules

Modules are the executable plugins that get the real job done. Usually modules can take "key=value" arguments and run in customized way depending up on the arguments themselves. A module can be invoked commandline or can be included in an Ansible playbook. We will discuss playbooks in a minute but for now, let us see the modules in action.

```
$ ansible -i inventory all -m ping
```

Above example will use the ping module to ping all the hosts defined in the inventory. There are several modules available in ansible. Let us try another one.

```
$ ansible -i inventory webservers -m command -a "ls"
```

In Above example, we used command module to fire ls command on the webservers group. We are now ready for remote execution. Adhoc tasks like restarting web servers or clearing out tmp files can be done easily using command module.

Ansible Roles

Ansible Roles, as the name suggests, define a particular role which a host or a group of hosts is going to serve. Typically, a host is associated with many roles. A single webserver can have the following roles associated with it:

- user management role
- basic configuration role
- webserver role.

Ansible Playbooks

Playbooks associate roles to the hosts. This is where we will tell Ansible to apply certain roles to a host or a group of hosts.

Not so good way to work with Ansible

Technically, it is possible to write a single yml file where we can define all the tasks and the host details effectively making it a mix of roles and playbook. However, that it is not a good idea to do so because:

- it makes the file diificult to read after a certain point
- it reduces code (essentially, roles) sharing

However, for the sake of learning, we will write the first yml file in simply like this:

```
---
- hosts: webserver
  tasks:
  - name: install httpd
    yum: pkg=httpd state=installed
```

The example above will install Apache web server on our systems. Let us also install some php libraries.

```
---
- hosts: webserver


  tasks:
  - name: install httpd
    yum: name=httpd state=installed


  - name: install php and libs
    yum: name={{ item }} state=installed
    with_items:
      - php-xml
      - php
      - php-zip
```

```
      - php-mbstring
      - php-mysqlnd
      - php-gd
```

Now we need to download the ownCloud app archive and extract it. We may need to do some more house keeping tasks.

Our final yml will look something like this:

```
---
- hosts: webserver

  tasks:
  - name: install httpd
    yum: name=httpd state=installed

  - name: install php and libs
    yum: name={{ item }} state=installed
    with_items:
      - php-xml
      - php
      - php-zip
      - php-mbstring
      - php-mysqlnd
      - php-gd

  - name: install additional tools
    yum: name=bzip2 state=installed

  - name: set selinux permissive
    selinux: policy=targeted state=permissive
```

```
  - name: get the owncloud tar
    get_url: url=https://download.owncloud.org/community/owncloud-
9.0.1.tar.bz2 dest=/var/www/html

  - name: exract the tar
    unarchive: src=/var/www/html/owncloud-9.0.1.tar.bz2 dest=/var/www/html
copy=no owner=apache
    notify:
    - restart apache

  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

Good Way of Working with Ansible

In the yml file above, we have three different components:

- hosts
- tasks
- handlers

According to Ansible best practices, we should break out these components. The hosts details should be defined in the playbook. The playbook should call the required roles which is where the tasks will go. Lastly, handlers are tasks but are executed at the end and only if they are called by a task. Best practices dictates that handlers should be defined separately.

So for a good setup our directory structure should look like this:

```
inventory1
variables
playbooks/
```

```
    webserver.yml
    database.yml
roles/
    role1/
        tasks/
            main.yml
        handlers/
            main.yml
        templates/
            template.j2
        files/
            file1
        vars/
            main.yml
```

Understand this structure and try to build this from the ownCloud playbook. We have built the setup as per the best practices. Check it out on github.

Some Common Modules

Ansible is good at configuring servers. There are a whole bunch of modules to help us do that.

- Software installation: yum, apt, pip
- Code management: git
- Remote execution: shell, command
- Services management: service
- Selinux management: selinux
- User manamgement: user

Examples:

```
- name: install git
```

```
  yum: pkg=git state=installed

- name: start nginx
  service: name=nginx state=started

- name: put selinux to enforcing mode
  selinux: policy=targeted state=enforcing

- name: create the user
  user: name=aditya
```

Introduction to Ansible Tags

Sometimes, it is useful to invoke only parts of a role or a playbook. For example, you need to add a new user to the system but you don't want to run the whole role or playbook for that.

With Ansible, it is possible to do so by using something known as tags.

```
- name: install httpd
  yum: name=httpd state=installed

- name: install php and libs
  yum: name={{ item }} state=installed
  with_items:
    - php-xml
    - php
    - php-zip
    - php-mbstring
    - php-mysqlnd
    - php-gd
  tags:
```

```
    - php
```

We have put a tag `php` on the task that was installing php packages. Now we can invoke the relevant playbook to run the task that has the right tag.

```
$ ansible-playbook -i inventory -t php playbooks/owncloud.yml
```

A task can have multiple tags. And same tag can be applied to many tasks.

Variables in Ansible

There are times when we have a bunch of similar servers but they are not exactly the same. For example, consider webservers. They may all run a webserver and might have same set of users accounts and acls but they may vary slightly in configuration. For such scenarios, variables are very helpful. A variable name can only consist of letters, numbers, and underscores and should always start with a letter. Below is an example of a variable definition in a playbook.

```
- hosts: webservers
  vars:
    http_port: 80
    domain: owncloud
```

While variables can be used in defining task blocks, most widespread use of variables is in defining configurations. We can use variable and templates to dynamically generate configurations and deploy on wide range of servers.

Ansible provides several variables off the shelf. Some of the most common ones are:

- ansible_os_family
- ansibble_hostname
- ansible_distribution

A full list can be obtained by using the following command:

```
$ ansible -i ~/tmp/inventory all -m setup
```

Setting up custom facts

Ansble can read custom facts from a remote machine. These facts are communicated to during an Ansible run and can be utilized to run playbooks and tasks in conditional manner.

Ansible can read any INI or json style file created in directory `/etc/ansible/facts.d` and make them available under `ansible_local` fact.

```
[general]
fact1=myfact1
fact2=myfact2
```

Now we can access these facts normally. To check if these facts are available, we can check out the facts.

```
ansible -i ~/tmp/inventory all -m setup -a "filter=ansible_local"
```

Registering Variables

Ansible can register and save the output of a task in a variable. This variable can be used later to invoke tasks conditionally.

```
- hosts: web_servers
  tasks:
  - shell: exit 5
    register: result_var
    ignore_errors: True

  - shell: echo return was 5
    when: result_var.rc == 5
```

Templates

Templates help in generating configurations on the fly. Ansible uses Jinja2 templates which ends with .j2 extension. Based on variables, these templates fill the blanks and generate configurations. These configurations are then uploaded to the target server. Example template:

```
<VirtualHost *:{{ http_port }} >
        ServerAdmin aditya@adityapatawari.com
        ServerName {{ domain }}
        Options Indexes MultiViews FollowSymLinks
        DirectoryIndex index.php
        DocumentRoot /var/www/html/owncloud
        <Directory /var/www/html/owncloud>
        Options FollowSymLinks
        AllowOverride All
    </Directory>
</VirtualHost>
```

Now we can use template module to read variables and fit the correct values and then put the configuration at the right place.

Condition Handling

Conditionals helps us evaluate a variable and take some action on the basis of the outcome. We can execute some part of the role or playbook depending up on the definition of the variable.

Example:

```
---
- yum: package=httpd state=installed
  when: ansible_os_family == "RedHat"
```

```
- apt: package=apache2 state=installed
  when: ansible_os_family == "Debian"
```

This is very useful to write OS agnostic roles.

Managing Secrets using Ansible Vault

Many of the application setups and server configurations need secrets like keys and passwords. Typically we store Ansible roles and plays in a version control but these version controls have wider access than we can afford in terms of exposing the secrts.

Ansible provides an inbuilt mechanism to encrypt the sensitive information encoded in yml format. For example, we can have a file with variables which can hold the passwords and can be encrypted.

```
---
mysql_password: top_secret
```

To encrypt the file above, we can call ansible-vault command line.

```
ansible-vault encrypt secret.yml
```

An encrypted file would look like this:

```
$ cat secret.yml
$ANSIBLE_VAULT;1.1;AES256
643635633534376231613562386335386439633835343362643861613137336631636165303333
332
623266656437623331623361343166383331396630636134 0a333666346264303264656339316
435
376331343438633762363366363535356363393837356337306564313039643162343233646 46
339
```

```
313962636665366264 0a3064393237623633333563633935633733343433386133636233633465
265
383063653763613635346632636534393237633261653762346235373362313933 65
```

Ansible Pull Mode

Traditionally Ansible is run in "push" mode where Ansible playbooks are run from a well known machine or from local machine and this, in turn, makes changes on remote machines as per the inventory.

However, Ansible also has a "pull" mode where remote machines would locally pull Ansible code from a git repository and execute `<fqdn.yml>` or `local.yml`. We can also give a filename to override this behavior.

```
$ ansible-pull -U https://github.com/training-devops/ansible-pull-example -i
<path_to_inventory>
```

We should put the command above in cron to execute this periodically.

Ansible Galaxy

Ansible Galaxy is the Ansible hub for sharing and reusing Ansible roles. Thousands of roles written by community are available for reuse. These roles are open sourced. Keep in mind that before using these roles, read them and understand them. Let us install a role from Ansible Galaxy.

```
ansible-galaxy install training-devops.el-repos
```

Above is a simple role to install EPEL and Remi repos. To understand the role better, it is advisable to browse through the role source which is available at github page. Note how this role uses best practices.

Modules for Cloud and Docker

Ansible, not only supports setting up bare-metal and virtualized environments, it can actually go one step ahead and setup a cloud based environment with providers like Amazon Web Services, Digital Ocean, Google Cloud, Open Stack and many more. There are modules to manage Docker based setup as well. An exhaustive list of cloud modules can be found on Ansible's documentation.

Examples:

```
- name: data container
  docker:
    name: mydata
    image: busybox
    state: present
    volumes:
    - /data
- digital_ocean:
    state: present
    command: ssh
    name: my_ssh_key
    ssh_pub_key: 'ssh-rsa AAAA...'
    api_token: XXX
```

Building Ansible Modules

Ansible comes with a lot of modules working out of the box. However, at times, if we need to write a custom module, then we can do so. An Ansible module can be written in any programming language but it is easiest to write a module in Python. A full guide is available on Ansible's documentation. We will create a simple module to check the return code of a website.

We have created the module and the code is available at github. To create a module, we need to create a directory called `library` where the modules will be kept.
A `main()` function is mandatory and this is where the execution begins.

```
#!/usr/bin/python


import urllib2
from ansible.module_utils.basic import *


def main():
    module = AnsibleModule(argument_spec={"url": {"required": True, "type":
"str" }})
    response = urllib2.urlopen(module.params["url"])
    module.exit_json(changed=True, meta={"return_code":response.code})


if __name__ == '__main__':
    main()
```

The module above can be called in a playbook like this:

```
- name: what is the return code
  url_code: url=http://trainingdevops.com
```

Using Ansible with Terrafoam

Terraform is a great provisioning tool. Ansible is great to set up machines. What happens when we add both to the mix?

What is Ansible?

Ansible is a modern automation tool which makes your life easier by managing your servers for you. You just need to define the configuration in which you are interested and ansible will go ahead and do it for you, be it installing a package or configuring a

server application or even restarting a service. Ansible is always ready to manage your servers. Read more about Ansible and its installation.

What is Terrafoam?

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions. In true sense, Terraform enables infrastructure as code. It works well with providers like Amazon Web Services, Google Cloud, Digital Ocean, Microsoft Azure and a lot more. Since Terraform is an independent tool, it can manage cross-cloud dependencies.

How to setup Terraform?

Terraform and its plugins are distributed as binaries. It is downloadable, in zip format, fromhttps://www.terraform.io/downloads.html. Once downloaded, we have to extract it and make sure that we add the extracted location in `$PATH` variable. Typically, doing this in `~/.bashrc` is the best way to manage `$PATH`. We recommend extracting it in /opt/terraform and then adding the line below in`~/.bashrc`

```
export PATH=$PATH:/opt/terraform
```

Important Terraform Components

Terraform consist of several components. Three of the important ones are listed below.

- **Resource**: It is a component that has to exist in the infrastructure like a compute instance or a DNS entry.
- **Provider**: Provider is the infrastructure provider like AWS, Google Cloud or Digital Ocean.
- **Provisioner**: Provisioners come in action after the resource is created successfully at the provider's end. Provisioner can execute a local command, invoke chef or execute a remote commend. We can use this to call Ansible.

Setting up ownCloud using Terraform and Ansible

We will run a virtual machine on Digital Ocean (they call it a droplet) using Terraform. For this, we need to generate an API token from Digital Ocean's website. So here, our provider is Digital Ocean and resource is the droplet. We will call Ansible locally using `local-exec` provisioner to set up ownCloud on the Droplet. Let us create a directory where we will keep all the files.

```
$ mkdir OConDO
$ cd OConCO
```

Terrafoam loads all the files in the current directory which ends with `.tf`. We will create two files,`credentials.tf` and `DO.tf`. We don't really need two files but it is a good idea to keep credentials in a different file. So, credentials.tf would look like this:

```
provider "digitalocean" {
    token =
"r514d6331b68dcg8fc9c0h83b792447cb43fvedz8fbc4205276g1c8v76s7925b"
}
```

Next, the DO.tf should look like this:

```
resource "digitalocean_droplet" "web" {
    image = "fedora-22-x64"
    name = "web-1"
    region = "sgp1"
    size = "512mb"
    ssh_keys = ["a1:b4:4d:4c:8d:8a:a3:de:41:94:b8:a0:f4:6f:8e:41"]

  provisioner "local-exec" {
    command = "sleep 30 && echo -e
\"[webserver]\n${digitalocean_droplet.web.ipv4_address}
```

```
ansible_connection=ssh ansible_ssh_user=root\" > inventory &&  ansible-
playbook -i inventory oc-playbook.yml"
  }
}
```

We have declared the resource to use the Fedora 22 image. We want a 512 mb droplet in Singapore region. We will name it as web-1. An additional parameter to let add my ssh key to the droplet has been added.

1. sleep for 30 seconds because sometimes it takes time for the droplet to allow ssh immediately.
2. add the IP address of the droplet with additional parameters to inventory file.
3. call Ansible with the inventory file.

We have a playbook ready to do the installation. The playbook, named oc-playbook.yml should look like this:

```
---
- hosts: webserver

  tasks:
  - name: install httpd
    dnf: name=httpd state=installed

  - name: install php and libs
    dnf: name={{ item }} state=installed
    with_items:
      - php-xml
      - php
      - php-zip
      - php-mbstring
      - php-mysqlnd
```

```
      - php-gd

  - name: install additional tools
    dnf: name=dnf state=installed

  - name: set selinux permissive
    selinux: policy=targeted state=permissive

  - name: get the owncloud tar
    get_url: url=https://download.owncloud.org/community/owncloud-
9.0.1.tar.bz2 dest=/var/www/html

  - name: exract the tar
    unarchive: src=/var/www/html/owncloud-9.0.1.tar.bz2 dest=/var/www/html
copy=no owner=apache
    notify:
    - restart apache

  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

Now, we have the Terraform artifacts and Ansible playbook ready, let us do a dry-run and see what all will be changed by Terraform.

```
terraform plan
+ digitalocean_droplet.web
    image:                "" => "fedora-22-x64"
    ipv4_address:         "" => "<computed>"
    ipv4_address_private: "" => "<computed>"
    ipv6_address:         "" => "<computed>"
    ipv6_address_private: "" => "<computed>"
```

```
    locked:                "" => "<computed>"
    name:                  "" => "web-1"
    region:                "" => "sgp1"
    size:                  "" => "512mb"
    ssh_keys.#:            "" => "1"
    ssh_keys.0:            "" =>
"a1:b4:4d:4c:8d:8a:a3:de:41:94:b8:a0:f4:6f:8e:41"
    status:                "" => "<computed>"


Plan: 1 to add, 0 to change, 0 to destroy.
```

We can verify that Terraform is going to create only one Digital Ocean droplet. There are no changes to the existing droplets and there is nothing to be destroyed. Once we have verified the Terraform output, we can simply call `terraform apply` and once the execution is finished, we can open the browser and checkout `http://<droplet_ip>/owncloud`.