

## Task 1 - Theory Overview

### Git

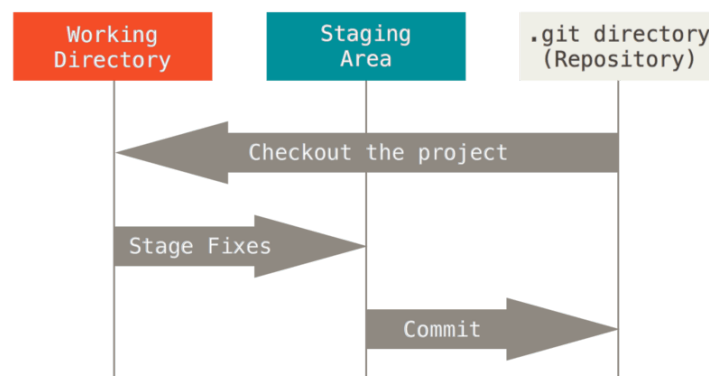
For your reference only, [here](#) is the Git downloads page, with [this article](#) being specific to the Linux/Unix systems, which we're utilizing in this lab. Therefore, you'd normally install Git with an `apt-get install git` for Debian/Ubuntu machines.

We can start working with the Git repositories in two ways. The first one is when you have some local directory on your system which you want to transform into a Git repository, which you would do with a `git init` command. This way, a new subdirectory named `.git` would be created within your directory, and in that `.git` subdirectory all the files necessary for tracking of changes, configuration and others would be stored. The second way is by an operation which Git calls this "cloning". This copies all files from the remote repository, along with the history of changes made.

To track the changes being made to a repository, Git utilizes a concept of commits. A commit will include an ID, a date, and some message, usually describing the change being made for readability. What it will also include is the identity of the person making said change. Therefore, before you're going to work with a remote Git repository, it is a good practice to configure the credentials for your remote Git user account, so that you will be able to commit the changes made on your repository.

When committing a file, we tell Git to store the data in our local repository database, which resides in the `.git` directory. This way, Git keeps track of the changes made to a file. We're able to easily verify the state of each file during each commit, thus we can easily revert our changes if needed, by restoring to previous version of the file.

Git consists of three main areas: working directory, staging area, and the beforementioned `.git` directory. The working directory, also called working tree, is rather self-explanatory.



A staging area contains the files that will be added to your repository database (which keeps track of the changes made to the files) upon next commit. Be aware that any files that weren't put into the staging area will not be committed.

Then, within a repository, Git separates the files between tracked and untracked. Git does that to know for which files to keep track of the changes being made.

A project in Gitlab is mainly where you will work on your code repository, but it offers way more than just the code. Within a project you'll be able to work on specific issues with your codebase, merge different branches or run the CI/CD pipelines, among many others.

## Ansible

ACI allows for automation via numerous ways – you can achieve that by using Python libraries such as acitoolkit, Ansible, Terraform, or simply with REST API calls, which you can leverage with most of programming languages (e.g., requests library for Python) or with tools like Postman. In this step we shall start working on implementing Infrastructure as Code (IaC) concepts using Ansible in our environment. One of the core concepts in DevOps is the automation of the IT Infrastructure wherever it is possible.

Ansible is an automation engine developed by Red Hat. What makes it stand out is the fact that you don't need to install any software on the IT Infrastructure devices – Ansible is agentless. It relies on Python to connect to the devices you specify and configure them in a simplified, codified fashion – using YAML files for that process. Another benefit of using Ansible is its adoption within the community – while there are many companies that are writing their own Ansible modules for their products, Red Hat allows the users to work on their own, custom modules, which are then put into a community collection. You'll see this firsthand when working on the VMware side of our lab.

All in all, you're able to easily automate your IT infrastructure, whether it's Cisco networking, VMware virtualization, public cloud provisioning, configuration management, application deployment, or anything else you might think of.

Ansible utilizes tasks to configure the devices. Said tasks can be run in two ways – either via an ad-hoc command or via file called a playbook. The main difference between the two is that an ad-hoc command will allow you to run a single task, while a playbook is a collection of tasks, making them reusable.

Ansible's ad-hoc command syntax looks as follows:

***ansible [pattern] -m [module]***

Modules in Ansible are groupings of code for particular use cases. There's a wide variety of both modules that are built into Ansible, as well as modules created by third party companies and the community. We won't be going into details on what are the uses for

each of these modules, but [here's](#) a list of all the built-in modules that come with Ansible for your reference.

Ansible modules are grouped together in what is called a collection since Ansible 2.10. You can find all the collections that can be used with Ansible by going to the Ansible Galaxy. It is there where you can find collections made available both by the third-party vendors and by the community. The page for it can be found [here](#).

Regarding patterns, these specify which remote nodes do you want to run your task on. And for your reference, here's a [link](#) describing patterns in more details.

The inventory is a definition of hosts running in your IT infrastructure. By default, it is found in `/etc/ansible/hosts` and commonly uses either INI or YAML file formats. A sample INI syntax can look like this:

```
server1.example.com
server2.example.com
10.0.0.1

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

Here's how that same sample inventory file would look like in a YAML syntax:

```
all:
  hosts:
    server1.example.com
    server2.example.com
    10.0.0.1
  children:
    db:
      hosts:
        db1.example.com
        db2.example.com
```

As you can probably deduce by now, the inventory allows you to provide details of your nodes by using both the hostnames and the IP addresses. Additionally, you can group hosts together, usually in accordance with their use within your environment. In our example, we've grouped the database servers together using the `'db'` group, which we can then refer to with a matching pattern in an ad-hoc command or in the playbook. You can read more on the inventory file components [here](#) or by running `less /etc/ansible/hosts`.

If you do not want to change your `hosts` inventory file, you can create a separate, project-specific inventory file and pass it to an ad-hoc command by using the `-i` option. This would be the syntax:

**`ansible [pattern] -i [inventory file location] -m [module]`**

Ansible has a concept of roles. Roles are basically meant to help you reuse the code for processes that will be repeatable in your environment based on particular use cases. Thus, you'd be specifying tasks, variables, files and other Ansible constructs and grouping

them together in a reusable role. Roles have their own directory structure, about which you can read more in the following resource:

- [Ansible Docs - Roles](#)

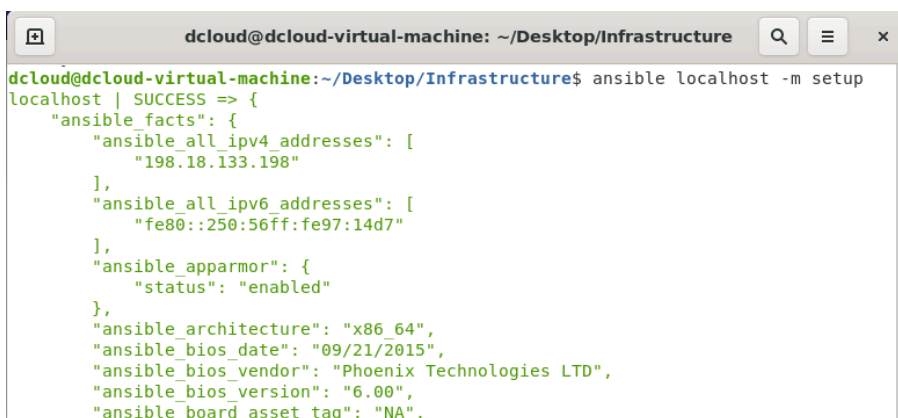
Then, there's the concept of variables. Ansible is very flexible when it comes to variables – you can read more on the variables themselves and on the way in which certain types of variables precede over another here:

- [Ansible Docs - Using variables](#)
- [Ansible Docs - Variable precedence](#)

If you want to play around with Ansible to grasp its basics, here are some tasks to follow. Keep in mind, these are completely optional.

1. Let's start by running an ad-hoc command to gather facts about our local machine:

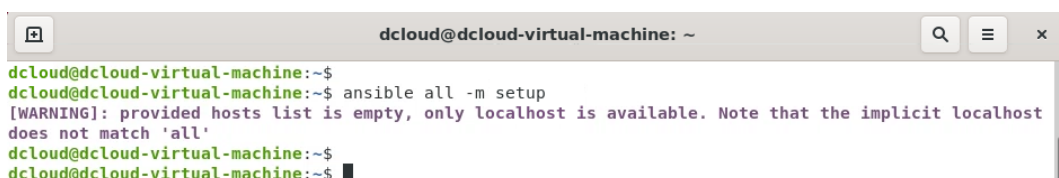
```
ansible localhost -m setup
```



```
dcloud@dcloud-virtual-machine: ~/Desktop/Infrastructure
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure$ ansible localhost -m setup
localhost | SUCCESS => {
  "ansible_facts": {
    "ansible_all_ipv4_addresses": [
      "198.18.133.198"
    ],
    "ansible_all_ipv6_addresses": [
      "fe80::250:56ff:fe97:14d7"
    ],
    "ansible_apparmor": {
      "status": "enabled"
    },
    "ansible_architecture": "x86_64",
    "ansible_bios_date": "09/21/2015",
    "ansible_bios_vendor": "Phoenix Technologies LTD",
    "ansible_bios_version": "6.00",
    "ansible_board_asset_tag": "NA",
```

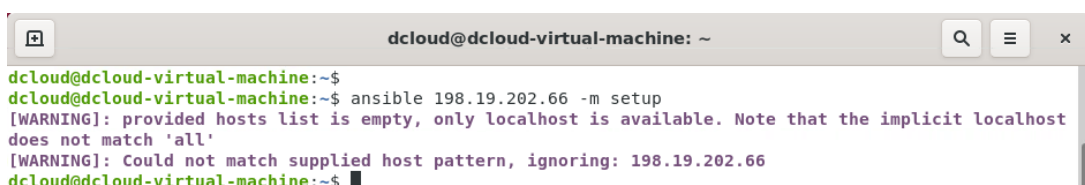
2. Test the ad-hoc command pattern by running the following:

```
ansible all -m setup
```



```
dcloud@dcloud-virtual-machine: ~
dcloud@dcloud-virtual-machine:~$
dcloud@dcloud-virtual-machine:~$ ansible all -m setup
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost
does not match 'all'
dcloud@dcloud-virtual-machine:~$
dcloud@dcloud-virtual-machine:~$
```

What you'll notice is an error mentioning an empty list of hosts. Similar issue would happen if you tried to use an IP or a hostname within the pattern.



```
dcloud@dcloud-virtual-machine: ~
dcloud@dcloud-virtual-machine:~$
dcloud@dcloud-virtual-machine:~$ ansible 198.19.202.66 -m setup
[WARNING]: provided hosts list is empty, only localhost is available. Note that the implicit localhost
does not match 'all'
[WARNING]: Could not match supplied host pattern, ignoring: 198.19.202.66
dcloud@dcloud-virtual-machine:~$
```

Thus, we need to start working on an inventory file.

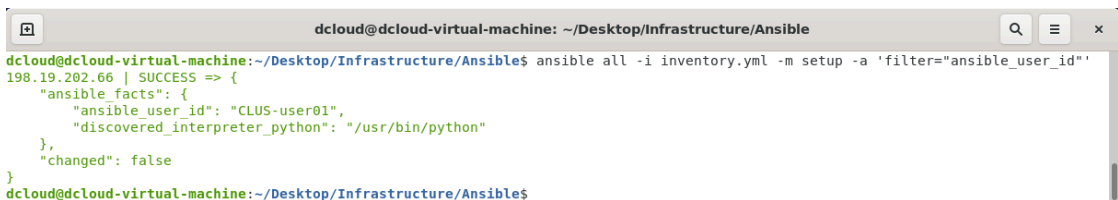
3. Run the following command:

```
ansible all -i inventory.yml -m setup
```

As you might've noticed, the output for this command is long. Thankfully, with ad-hoc Ansible commands, you can pass attributes used by the module as an option, by using the `-a` keyword.

Now, when you run the following command:

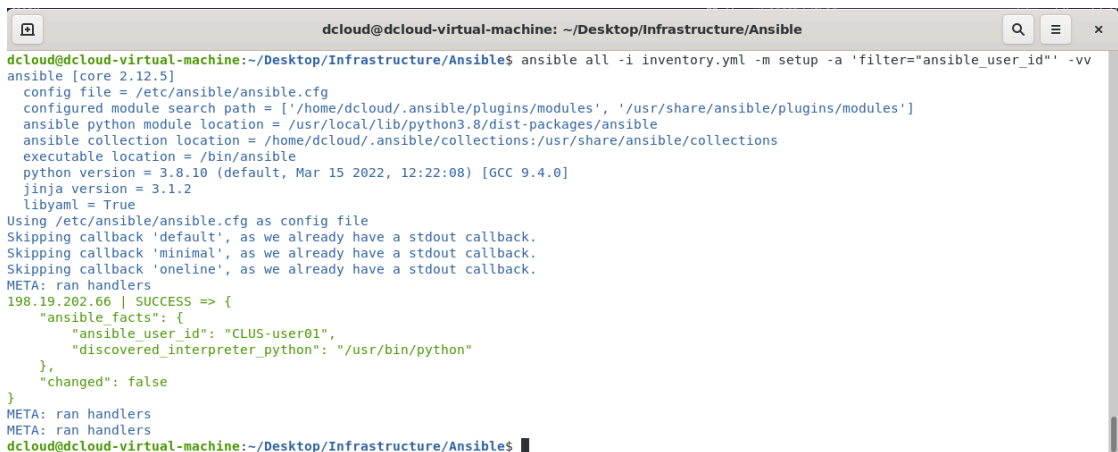
```
ansible all -i inventory.yml -m setup -a 'filter="ansible_user_id"'
```



```
dcloud@dcloud-virtual-machine: ~/Desktop/Infrastructure/Ansible
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure/Ansible$ ansible all -i inventory.yml -m setup -a 'filter="ansible_user_id"'
198.19.202.66 | SUCCESS => {
  "ansible_facts": {
    "ansible_user_id": "CLUS-user01",
    "discovered_interpreter_python": "/usr/bin/python"
  },
  "changed": false
}
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure/Ansible$
```

The output has been severely limited due to our usage of the `filter` attribute of the built-in `setup` module. Now, to check what attributes are available for each module, you should use Ansible documentation. For example, for the `setup` module, [this](#) is the page with the documentation.

Finally, you can run the ad-hoc commands with different verbosity levels, to see what's happening behind the scenes as Ansible runs the tasks. You'd do that by using the `-v` option. You can use from one to four v's (from `-v` to `-vvvv`) to increase/decrease the level of verbosity. Try using our previous command with different levels of verbosity.



```
dcloud@dcloud-virtual-machine: ~/Desktop/Infrastructure/Ansible
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure/Ansible$ ansible all -i inventory.yml -m setup -a 'filter="ansible_user_id"' -vv
ansible [core 2.12.5]
  config file = /etc/ansible/ansible.cfg
  configured module search path = ['/home/dcloud/.ansible/plugins/modules', '/usr/share/ansible/plugins/modules']
  ansible python module location = /usr/local/lib/python3.8/dist-packages/ansible
  ansible collection location = /home/dcloud/.ansible/ansible/collections:usr/share/ansible/collections
  executable location = /bin/ansible
  python version = 3.8.10 (default, Mar 15 2022, 12:22:08) [GCC 9.4.0]
  jinja version = 3.1.2
  libyaml = True
Using /etc/ansible/ansible.cfg as config file
Skipping callback 'default', as we already have a stdout callback.
Skipping callback 'minimal', as we already have a stdout callback.
Skipping callback 'oneline', as we already have a stdout callback.
META: ran handlers
198.19.202.66 | SUCCESS => {
  "ansible_facts": {
    "ansible_user_id": "CLUS-user01",
    "discovered_interpreter_python": "/usr/bin/python"
  },
  "changed": false
}
META: ran handlers
META: ran handlers
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure/Ansible$
```

4. After playing around with the ad-hoc commands it's easy to see what some of the limitations of this method are. An ad-hoc command is fine for testing, or for running automation once (tasks like server reboot for example), but even then, if we want

to use multiple attributes for a module that we're using in our task, we'd have to specify each one with a separate `-a` option. Instead, let's create our own playbook.

In this case, let us just test the connectivity towards an APIC present in our on-premises site and grab some outputs regarding the APIC interface, with finally collecting some details about the Eth2/1 interface of said APIC. Please create a .YML file for the playbook and use the following code:

```
- name: "Verifying connectivity to the APIC and gathering facts"
  hosts: 198.19.202.66

  tasks:
    - name: "Using the ping module towards the APIC"
      ansible.builtin.ping:

    - name: "Gathering facts about the Ansible env on the APIC"
      ansible.builtin.setup:
        filter:
          - 'ansible_env'
      register: command_output

    - name: "Printing the output of the Ansible env on the APIC"
      debug:
        var: command_output

    - name: "Getting the list of interfaces on the APIC"
      ansible.builtin.setup:
        filter:
          - 'ansible_interfaces'
      register: command_output

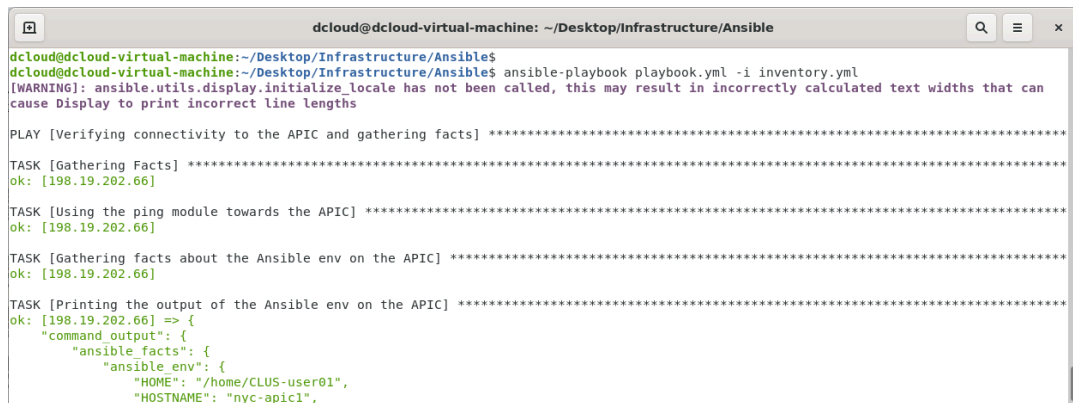
    - name: "Printing the list of the interfaces on the APIC"
      debug:
        var: command_output

    - name: "Gathering facts about one of the NICs on the APIC"
      ansible.builtin.setup:
        filter:
          - 'ansible_eth2_1'
      register: command_output

    - name: "Printing the facts about one of the NICs on the APIC"
      debug:
        var: command_output
```

5. Run the playbook created above by running the following command:

```
ansible-playbook playbook.yml -i inventory.yml
```

A terminal window titled 'dcloud@dcloud-virtual-machine: ~/Desktop/Infrastructure/Ansible'. The prompt is 'dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure/Ansible\$'. The command entered is 'ansible-playbook playbook.yml -i inventory.yml'. The output shows a warning about locale, followed by a play 'Verifying connectivity to the APIC and gathering facts'. Tasks include 'Gathering Facts', 'Using the ping module towards the APIC', 'Gathering facts about the Ansible env on the APIC', and 'Printing the output of the Ansible env on the APIC'. The final output is a JSON object showing 'ansible\_facts' with 'ansible\_env' containing 'HOME' and 'HOSTNAME' values.

```
dcloud@dcloud-virtual-machine:~/Desktop/Infrastructure/Ansible$ ansible-playbook playbook.yml -i inventory.yml
[WARNING]: ansible.utils.display.initialize_locale has not been called, this may result in incorrectly calculated text widths that can
cause Display to print incorrect line lengths

PLAY [Verifying connectivity to the APIC and gathering facts] *****

TASK [Gathering Facts] *****
ok: [198.19.202.66]

TASK [Using the ping module towards the APIC] *****
ok: [198.19.202.66]

TASK [Gathering facts about the Ansible env on the APIC] *****
ok: [198.19.202.66]

TASK [Printing the output of the Ansible env on the APIC] *****
ok: [198.19.202.66] => {
  "command_output": {
    "ansible_facts": {
      "ansible_env": {
        "HOME": "/home/CLUS-user01",
        "HOSTNAME": "nyc-apic1",
```

## Task 2 - Theory Overview

Git uses a concept of branches. Branching means you diverge from the main line of development and continue to do work without messing with that main line.

How do branches work? Git doesn't store data as a series of changesets or differences, but instead as a series of *snapshots*. When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it. A branch in Git is simply a lightweight movable pointer to one of these commits.

There are different resources available that describe the Git workflows across branches that you can refer to, [for example this one](#). A popular one is to have separate branches for development, staging and production.

## Task 3 - Theory Overview

For this first task we will go with Terraform as it is a very easy to code in but equally powerful.

Terraform is an automation solution created by Hashicorp written in Go language (you will not need to know Go unless you want to create your own providers). We will be writing in HCL Language which is extremely close to JSON.

Generally, when we use Terraform, we have 3 stages in the pipeline- Validate, Plan and Apply. For each of these, terraform has inbuilt commands (namely terraform validate, terraform plan and terraform apply) that can help us progress in the pipeline.

**Terraform init** automatically understands and downloads the dependencies that the script requires and gets it ready for execution. These are providers mainly, that we will talk about shortly. This is our step 0 that we need to do before we start with the validate stage of the pipeline. Upon running init, a hidden .terraform folder with all the necessary dependencies gets downloaded and created. You will also see a .terraform.lock.hcl file created – this file records all the provider selections that you made for this code, which in turn can guarantee that Terraform will make the same selections when you run the code in the future.

**Terraform validate** is like a config check that will verify if our scripts have no syntax/declaration issues. It does not do any checks for the remote services (such as MSO/AWS) here.

**Terraform plan** command performs a terraform syntax check of your configuration files, creates an execution plan, which lets you preview the changes that Terraform plans to make to your infrastructure. At this stage terraform will contact the remote services and understand what changes are to be made, whether there are any hindrances to performing the intended changes (for example an existing object with the same name, a dependency that isn't present already or a reachability issue to the APIC/AWS).

Terraform plan also resolves or identifies any object dependencies. For example, a tenant must be available or created before its child VRF is created. These Dependencies are automatically identified by Terraform, which is what makes it such a powerful tool.

At the end of the execution of Terraform plan, you will know the exact objects that are going to be created, modified, and deleted. You could call this stage a 'dry run' of our configuration.

**Terraform apply** is the actual application of the plan. By default, terraform apply runs a plan first automatically and then proceeds with making the changes.

Each of these will get clearer as we create our script in the next task. One of the main features of Terraform is it is that it keeps state, unlike Ansible. If you use Ansible to create playbooks, you will have to guide it to perform certain specific configuration by mentioning how Ansible is to set things up and what order resources are built. In Terraform you do not do manual configurations – just tell it what your desired final state is.

The following example makes this very clear. Suppose in our script where we create an EPG and define some parameters, we change the name of an EPG and then re-execute it. Ansible is going to create another EPG instance with the new name. Terraform however saves state and hence is going to rename the EPG that was referenced with the old one as it knows it was changed.

There are 2 kinds of objects in Terraform – **Data Sources** and **Resources**.



**Data sources** allow data to be fetched or computed for use elsewhere in Terraform configuration. Use of data sources allows a Terraform configuration to build on information defined outside of Terraform or defined by another separate Terraform configuration. The provider is responsible for defining and maintaining the data source. For example, a *'mso\_site'* data source contains information such as the site name and site id which can be used in other parts of the Terraform code to identify/modify these values.

**Resources** are a component of our infrastructure. It may be a small component like a BD or an EPG or even a Tenant or Schema. This is the most important component of our configuration, as we will be using Resources to create new objects and modify our setup.

## Task 5 - Theory Overview

### EC2

An EC2 can be visualized as a Virtual Machine running on the cloud which can host whatever services we are interested in.

**Security groups** An AWS security group acts as an 'ACL'(Access List) for your EC2 instances to control incoming and outgoing traffic. Both inbound and outbound rules control the flow of traffic to and traffic from your instance, respectively.

A route table contains a set of rules, called routes, that determine where network traffic from your subnet or gateway is directed.

On the remote-exec, we add a trigger called `build_number= timestamp()` and hence it will execute every single time you terraform apply and not just the first time you apply. You could modify this behavior by removing this. The default behavior of a provisioner is to only execute on the first terraform apply.