Ansible

Raúl Estrada

Octubre 2020

Ansible

In this section, we are going to take our first steps toward a more comprehensive example in Ansible. For now, we are going to install and configure NGINX, a very popular web server so we can showcase the main concepts of Ansible.

First, we are going to create a VM in Google Cloud Platform with an associated static IP so we can target it from our inventory. We are going to use Terraform in order to do it. First, we'll look at our resources file:

Copy

```
provider "google" {
  credentials = "${file("account.json")}"
  project = "${var.project_name}"
  region = "${var.default_region}"
}

resource "google_compute_instance"
"nginx" {
  name = "nginx"
  machine_type = "n1-standard-1"
  zone = "europe-west1-b"
  disk {
   image = "ubuntu-os-cloud/ubuntu-1704-zesty-v20170413"
  }
  network_interface {
    network = "default"
    access_config {
      nat_ip = "${google_compute_address.nginx-ip.address}"
    }
  }
```

And now, we'll look at our vars file:

Copy

```
variable "project_name" {
  type = "string"
  default = "implementing-modern-devops"
}

variable "default_region" {
  type = "string"
  default = "europe-west1"
}
```

In this case, we are reusing the project from the previous chapter as it is convenient to shut down everything once we are done. Now we run our plan so we can see what resources are going to be created:

Copy

```
+ google_compute_address.nginx-ip
  address: "<computed>"
  name: "nginx-ip"
  self_link: "<computed>"

+ google_compute_instance.nginx
  can_ip_forward: "false"
  disk.#: "1"
  disk.0.auto_delete: "true"
  disk.0.image: "ubuntu-os-cloud/ubuntu-1704-zesty-v20170413"
  machine_type: "n1-standard-1"
  metadata_fingerprint: "<computed>"
  name: "nginx"
  network_interface.#: "1"
  network_interface.0.access_config.#: "1"
  network_interface.0.access_config.0.assigned_nat_ip: "<computed>"
  network_interface.0.access_config.0.nat_ip: "<computed>"
  network_interface.0.address: "<computed>"
  network_interface.0.name: "<computed>"
  network_interface.0.network: "default"
```

So far, everything looks right. We are creating two resources:

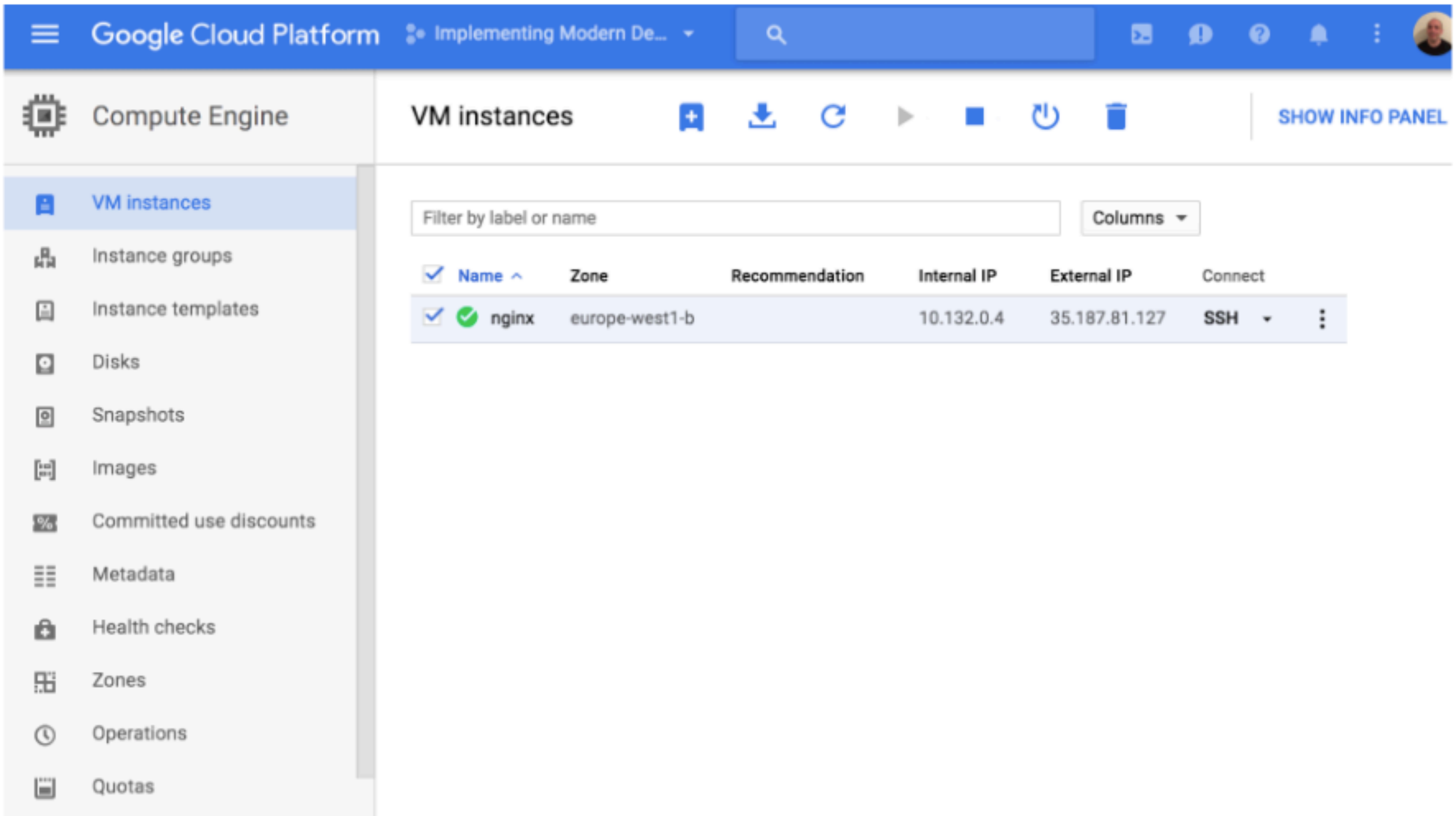
- The static IP
- The VM

Now, we can apply our infrastructure:

Copy

```
google_compute_address.nginx-ip: Creating...
 address: "" => "<computed>"
 name: "" => "nginx-ip"
 self_link: "" => "<computed>"
google_compute_address.nginx-ip: Still creating... (10s elapsed)
google_compute_address.nginx-ip: Creation complete
google_compute_instance.nginx: Creating...
 can_ip_forward: "" => "false"
 disk.#: "" => "1"
 disk.0.auto_delete: "" => "true"
 disk.0.image: "" => "ubuntu-os-cloud/ubuntu-1704-zesty-v20170413"
 machine_type: "" => "n1-standard-1"
 metadata_fingerprint: "" => "<computed>"
 name: "" => "nginx"
 network_interface.#: "" => "1"
 network_interface.0.access_config.#: "" => "1"
 network_interface.0.access_config.0.assigned_nat_ip: "" => "<computed>"
 network_interface.0.access_config.0.nat_ip: "" => "35.187.81.127"
 network_interface.0.address: "" => "<computed>"
 network_interface.0.name: "" => "<computed>"
```

And everything works as expected. If we check Google Cloud Platform, we can see that our VM has been created and has associated a public IP:



In this case, the associated public IP is `35.187.81.127` . It is important to verify that we can reach the server via SSH. In order to do it, just click on the **`SSH`** button on the right-hand side of your instance row and it should open a Cloud Console window with terminal access.

Once everything is up and running, it is time to start with Ansible. First, we are going to create our inventory file:

Copy

```
[nginx-servers]
35.187.81.127
```

This is very simple: a group with our public IP address that is connected to our VM. Save the file with the name `inventory` in a new folder named, for example, `ansible-nginx`. Once the inventory is created, we need to verify that all the hosts can be reached. Ansible provides you the tool to do that:

Copy

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

If you execute the preceding command, Ansible will `ping` (actually, it does not use the ping command but tries to issue a connection to the server) all the hosts in your inventory specified in the parameter `-i`. If you change everything for the name of a group, Ansible will try to reach only the hosts in that group.

Let's take a look at the output of the command:

Copy

```
35.187.81.127 | UNREACHABLE! => {
 "changed": false,
 "msg": "Failed to connect to the host via ssh: Permission denied (publickey).\r\n",
 "unreachable": true
}
```

We are experiencing problems in connecting to our remote host and the cause is that we don't have any key that the host can validate to verify our identity. This is expected as we did not configure it, but now, we are going to solve it by creating a key pair and installing it on the remote host using the Google Cloud SDK:

Copy

```
gcloud compute ssh nginx
```

This command will do three things:

> - Generate a new key pair
> - Install the key pair in our remote VM
> - Open a shell in our VM in GCP

The new key generated can be found under `~/.ssh/` with the name `google_compute_engine and google_compute_engine.pub` (private and public key).

Once the command finishes, our shell should look like this:

```
Welcome to Ubuntu 17.04 (GNU/Linux 4.10.0-19-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

 * Ubuntu 12.04 LTS ('precise') end-of-life was April 28, 2017
   ongoing security updates for 12.04 are available with Ubuntu Advantage
   - https://ubu.one/U1204esm
 * Aaron Honeycutt from the Kubuntu Council on art and design in Kubuntu
   - https://ubu.one/kubuart
 * The Ubuntu Desktop team wants your feedback on the move to Gnome
   - https://ubu.one/2GNome

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

0 packages can be updated.
0 updates are security updates.


Last login: Fri May 26 01:25:24 2017 from 79.97.8.5
davidgonzalez@nginx:~$
```

Now we have a terminal connected to our VM and we can execute commands. `gcloud` configures a user by default; in my case, `davidgonzalez` that can use `sudo` without password. In this case, we are going to execute the playbook as the root, so we need to be able to login as root into the VM. Copy the file `~/.ssh/authorized_keys` into `/root/.ssh/authorized_keys` and we should be able to do it. So, we have copied the public key that we generated earlier to the set of authorized keys of the root user.

> **Note**
>
> In general, root access should be avoided as much as possible, but in this case, we will be executing the playbook as the root for convenience.

In order for Ansible to be able to use the key, we need to add it to the daemon on our server:

Copy

```
ssh-add ~/.ssh/google_compute_engine
```

This command should output the success, stating that the identity was added.

Now we can run our pin command again:

Copy

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

The output should be very different:

Copy

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

This means that now, Ansible is able to reach our server; therefore, it will be able to execute the playbook against it.

Now it is time to start writing our first `ansible` playbook. Inside the same folder, `ansible-nginx` , create a file called `tasks.yml` with the following content:

Copy

```
---
- hosts: all
 user: root
 tasks:
 - name: Update sources
   apt:
     update_cache: yes
 - name: Upgrade all packages
   apt:
     upgrade: dist
```

This is simple to understand:

- Our playbook is going to affect all the hosts
- The user running the playbook is going to be root
- And then we are going to execute two tasks:
  - Update the `apt cache`
  - Upgrade all the packages

Once we have the two files (inventory and playbook), we can run the following command:

Copy

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

We should produce output similar to the following one:

```
PLAY [all] ********************************************

TASK [setup] ******************************************
ok: [35.187.81.127]

TASK [Update sources] *********************************
changed: [35.187.81.127]

TASK [Upgrade all packages] ***************************
changed: [35.187.81.127]

PLAY RECAP ********************************************
35.187.81.127 : ok=3 changed=2 unreachable=0 failed=0
```

Let's explain the output:

> - First, it specifies against which group we are going to execute the playbook. In this case, we specified that the group is `all` .
> - Then, we can see three tasks being executed. As you can see, the description matches the description specified in `tasks.yml` . This is very helpful in order to understand the output of your playbooks, especially when they fail.
> - And then we get a recap:
>
>   > - Three tasks were executed
>   > - Two of them produced changes on the server
>   > - Zero failed

Simple and effective. This is the closest to executing a script in the server that we can get: a set of instructions, a target host, and its output.

In Ansible, instead of plain bash instructions, the actions are encapsulated into modules. A module is a component of the DSL, which allows you to do something special. In the playbook from earlier, apt is a module included in the core of Ansible. Documentation for it can be found at http://docs.ansible.com/ansible/apt_module.html.

Let's take another look to one of our usages of the `apt` module:

Copy

```
- name: Update sources
  apt:
    update_cache: yes
```

This, as you can guess, would be the equivalent to the following:

Copy

```
apt-cache update
```

So, in this case, Ansible provide us with a different module called command, which allows us to execute commands in the hosts of our inventory. Take a look at the following `yaml` :

Copy

```
- name: Update sources
  command: apt-cache update
```

This is equivalent to the `yaml` from earlier, and both do the same: update `apt-cache` .

In general, if there is a module for a given task, it is recommended that you use it as it will handle (or at least you can expect it to) the errors and the outputs better than executing the equivalent command.

Now, once our playbook has succeeded, we can expect our system to be up to date. You can check it by running the playbook again:

```
PLAY [all] *******************************************

TASK [setup] *****************************************
ok: [35.187.81.127]

TASK [Update sources] ********************************
changed: [35.187.81.127]

TASK [Upgrade all packages] **************************
ok: [35.187.81.127]

PLAY RECAP *******************************************
35.187.81.127 : ok=3 changed=1 unreachable=0 failed=0
```

Now you can see that only one task has produced changes in the server (updating the apt sources).