

**TERRAFORM + ANSIBLE + INSPEC:
AUTOMATIZACIÓN Y CONTROL
DE LA
INFRAESTRUCTURA Y LA CONFIGURACIÓN**

**IES Gonzalo Nazareno
Proyecto Fin de Ciclo
Álvaro Beleño Rodríguez
05/06/2019**

Índice de contenidos

Introducción.....	4
Escenario.....	4
¿Qué es Terraform?.....	4
¿Qué es Microsoft Azure?.....	4
¿Qué es Ansible?.....	4
¿Qué es InSpec?.....	5
Instalación de las herramientas.....	5
Instalación de Terraform.....	5
Instalación de Ansible.....	5
Instalación mediante repositorio oficial.....	5
Instalación mediante pip.....	6
Instalación de InSpec.....	7
Instalación de Azure CLI.....	7
Primeros pasos.....	8
Autenticación en Azure para Terraform.....	8
Autenticación mediante la Azure CLI.....	8
Autenticación mediante un Service Principal y un certificado del cliente.....	8
Autenticación en Azure para Ansible.....	12
Autenticación mediante la Azure CLI.....	12
Autenticación mediante un fichero de credenciales.....	12
Autenticación mediante variables de entorno.....	13
Autenticación en Azure para InSpec.....	13
Autenticación mediante fichero de credenciales.....	14
Autenticación mediante variables de entorno.....	14
Anotaciones sobre los métodos de autenticación.....	14
Pruebas de autenticación.....	15
Terraform.....	15
Ansible.....	17
InSpec.....	18
Uso de submódulos en el repositorio.....	19
Creación de la infraestructura (I).....	19
Trabajando con módulos en Terraform.....	19
Módulos cuyo origen es un repositorio Git.....	20
Trabajando con Dynamic Inventory de Ansible.....	21
¿Qué es el Dynamic Inventory de Ansible?.....	21
¿Cómo se utiliza?.....	21
Primeros tests con InSpec.....	23
InSpec y los outputs de Terraform.....	25
¿Qué son los outputs de Terraform?.....	25
¿Cómo se definen?.....	26
Accediendo a la IP desde la Azure CLI.....	27
De “terraform output” a variables de entorno.....	27
Shift left testing.....	28
Creación de la infraestructura (II).....	28

Terraform.....	28
Inspec (Shift left testing).....	30
Ansible.....	31
Filtrado por nombre de las máquinas.....	32
Ejecutando los playbooks.....	32
InSpec.....	35
Generando las variables de entorno.....	35
Ejecutando los tests.....	36
Otros apuntes.....	38
Alternativas.....	38
Terraform.....	38
Azure.....	39
Ansible.....	39
InSpec.....	40
Siguientes pasos.....	40
Conclusiones.....	40
Webgrafía.....	42

Introducción

En el desarrollo de esta memoria, desplegaremos mediante Terraform un escenario compuesto de dos máquinas virtuales en Azure, en las cuales ejecutaremos recetas de Ansible que configurarán una base de datos PostgreSQL en una de las máquinas y un CMS (En este caso Drupal) sobre Apache en la otra. Finalmente, ejecutaremos InSpec para testear nuestra infraestructura y aplicaciones y así comprobar que cumplen con los requisitos que les imponemos en dichos tests.

La meta de este proyecto será utilizar las tres tecnologías anteriormente nombradas, ver cómo se integran tanto con Azure como entre ellas mismas y ver cuáles podrían ser las opciones que nos interesarían más a la hora de trabajar con estas herramientas.

Escenario

El desarrollo de este proyecto se realizará en *arca*, una máquina Debian 9 Stretch con 4 GB de memoria RAM. Mediante *arca*, desplegaremos con Terraform dos máquinas virtuales en Azure también Debian 9 Stretch cada una con 4 GB de memoria RAM, 2 cores y 8 GiB de espacio en disco.

¿Qué es Terraform?

Es una herramienta desarrollada por HashiCorp, que se encarga de la orquestación de la infraestructura, la cual definiremos mediante código de alto nivel, pudiendo ser en HCL (HashiCorp Configuration Language) o en JSON. Este código a su vez generará un plan de ejecución mediante el cual se desplegará nuestra infraestructura en el proveedor elegido, en nuestro caso *Microsoft Azure*.

¿Qué es Microsoft Azure?

Microsoft Azure es la plataforma *cloud* de Microsoft, un *IaaS (Infrastructure as a service)* desde el cual se gestiona la infraestructura que se nos ofrece para que podamos utilizar los recursos que necesitemos/queramos, pagando un precio determinado según los recursos y el tiempo que los utilicemos.

¿Qué es Ansible?

Ansible es una herramienta mediante la cual podremos aprovisionar equipos. En otras palabras, podremos gestionar automáticamente la configuración de los mismos. Iremos definiendo las tareas a ejecutar en ficheros YAML, que recibirán el nombre de *tasks*. Podemos agrupar estos *tasks* en *roles*. Finalmente, definiremos el orden en el que se ejecutarán dichos *roles* en un *playbook*.

¿Qué es InSpec?

InSpec es un *framework* encargado del *testing* tanto de aplicaciones como de infraestructura. Estos tests se hacen comparando el estado actual de la infraestructura (o la aplicación) con el estado deseado, que declararemos mediante código.

Instalación de las herramientas

Instalación de Terraform

Para instalar Terraform, iremos al apartado de Descargas de la página oficial de la herramienta (<https://www.terraform.io/downloads.html>) y descargaremos la versión correspondiente a nuestro sistema operativo y nuestra arquitectura (En mi caso Linux 64 bit).

Al hacer esto, obtendremos un archivo con extensión .zip, el cual descomprimiremos. Una vez hecho esto, tendremos el binario de Terraform, el cual moveremos al directorio `/usr/bin` y ya podremos ejecutar el comando `terraform`.

También podremos hacerlo teniendo el binario de Terraform en un directorio específico y añadiendo dicho directorio a nuestra variable de entorno PATH. Por ejemplo, si tenemos dicho binario en el directorio `/home/debian/terraform`, añadiríamos dicho directorio a la variable PATH de la siguiente manera:

```
export PATH=/home/debian/terraform:$PATH
```

Cuando hayamos hecho esto, ya tendríamos disponible para ejecutar nuevamente el comando `terraform`.

Para verificar que hemos instalado Terraform correctamente, podremos ejecutar simplemente el comando `terraform` y deberíamos ver un *output* similar al siguiente:

```
alvaro@arca:~$ terraform
Usage: terraform [-version] [-help] <command> [args]

The available commands for execution are listed below.
The most common, useful commands are shown first, followed by
less common or more advanced commands.
[...]
```

Instalación de Ansible

Instalación mediante repositorio oficial

La instalación de Ansible en Debian es muy sencilla.

Sólo tendremos que añadir la siguiente línea al fichero `/etc/apt/sources.list`:

```
deb http://ppa.launchpad.net/ansible/ansible/ubuntu trusty main
```

Una vez hecho esto, añadiremos la clave de firmado para poder, finalmente, instalar Ansible. Todo esto lo haremos mediante los siguientes comandos:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys
93C4A3FD7BB9C367
sudo apt-get update
sudo apt-get install ansible
```

Instalación mediante pip

También tenemos otra forma de instalar Ansible, y es mediante un entorno virtual de Python.

Primero, instalaremos el paquete *virtualenv*, que será lo que utilizaremos para crear dicho entorno virtual. Ahora, con *virtualenv* instalado, ejecutaremos el siguiente comando para crear un entorno virtual de Python 3:

```
virtualenv -p python3 <nombre del virtualenv>
# En mi caso:
virtualenv -p python3 ansible
```

Acto seguido, activaremos este entorno virtual de la siguiente manera:

```
source <ruta del virtualenv>/bin/activate
# En mi caso:
source ~/env/ansible/bin/activate
```

Teniendo esto, podremos instalar Ansible mediante *pip* con el siguiente comando:

```
pip install ansible
```

Y ya tendremos Ansible instalado en nuestro equipo.

Nota: Si instalamos Ansible mediante este método, tendremos que activar el entorno virtual cada vez que queramos utilizar dicha herramienta. Para desactivar dicho entorno virtual, utilizaremos el comando deactivate.

Con independencia del método de instalación que hayamos elegido, al ejecutar el comando *ansible*, deberíamos obtener una salida como la siguiente:

```
(ansible) alvaro@arca:~/env$ ansible --version
ansible 2.8.0
  config file = None
[...]
```

Instalación de InSpec

Para instalar InSpec, desde su página web iremos al apartado de Descargas y descargaremos la versión acorde con nuestro sistema operativo (En este caso, seleccionaremos la versión para Ubuntu). Con esto, obtendremos un fichero con extensión *.deb*, que podremos finalmente instalar mediante el siguiente comando:

```
sudo dpkg -i inspec*.deb
```

Tras hacer esto, ya dispondremos de InSpec instalado en nuestra máquina. Esto lo podremos comprobar de la siguiente manera:

```
alvaro@arca:~$ inspec --version
4.3.2
```

Instalación de Azure CLI

Para llevar a cabo la instalación de Azure CLI, primero tendremos que instalar los paquetes *curl*, *apt-transport-https*, *lsb-release* y *gpg*. Una vez teniendo estos paquetes instalados, tendremos que descargar la clave de firmado de Microsoft:

```
curl -sL https://packages.microsoft.com/keys/microsoft.asc | \
  gpg --dearmor | \
  sudo tee /etc/apt/trusted.gpg.d/microsoft.asc.gpg >
/dev/null
```

A continuación, agregaremos el repositorio desde el que descargaremos Azure CLI:

```
AZ_REPO=$(lsb_release -cs)
echo "deb [arch=amd64]
https://packages.microsoft.com/repos/azure-cli/ $AZ_REPO main"
| \
  sudo tee /etc/apt/sources.list.d/azure-cli.list
```

Y por último, actualizaremos la lista de paquetes y descargaremos *azure-cli*:

```
sudo apt-get update
sudo apt-get install azure-cli
```

Ya tendremos la utilidad de Azure de línea de comandos instalada, como podemos comprobar:

```
alvaro@arca:~$ az --version
azure-cli                2.0.64

acr                      2.2.6
[...]
```

Primeros pasos

Autenticación en Azure para Terraform

La autenticación con Azure para Terraform la podremos hacer de varias formas.

Autenticación mediante la Azure CLI

Podremos autenticarnos en Azure mediante la Azure CLI, lo que haremos mediante el siguiente comando:

```
az login [--use-device-code]
```

Cuando ejecutemos este comando, se nos proporcionará una URL y un código que usaremos para autenticarnos, con un *output* como el siguiente:

```
To sign in, use a web browser to open the page
https://microsoft.com/devicelogin and enter the code <código>
to authenticate.
```

Nota: Si ejecutamos el comando anterior sin --use-device-code, se nos abrirá directamente una pestaña en nuestro navegador para que podamos seleccionar la cuenta que queramos.

Entraremos en la URL y pegaremos el código que se nos ha proporcionado. Acto seguido, se nos pedirá que seleccionemos la cuenta con la que deseamos hacer *login*. Tras hacer esto, nos aparecerá como salida un JSON en el que nos aparecerá información sobre las suscripciones con las que contamos en la cuenta con la que hemos iniciado sesión.

Este *output* sería el mismo que si ejecutáramos el siguiente comando:

```
az account list
```

Y podríamos mostrarlo de una manera más visual de la siguiente manera:

```
az account list -o table
```

Autenticación mediante un Service Principal y un certificado del cliente

También podremos autenticarnos en Azure a través de un *Service Principal* y un certificado del cliente. Un *Service Principal* es básicamente una aplicación en el Azure Active Directory cuyos *tokens* de autenticación podremos usar como los campos *client_id*, *client_secret* y *tenant_id* necesarios para la autenticación de Terraform.

Primero, tendremos que crear el certificado del cliente, que se utilizará para la autenticación del mismo. Para esto, crearemos antes de nada un fichero *.csr* con el siguiente comando:


```
openssl req -newkey rsa:4096 -nodes -keyout "service-
principal.key" -out "service-principal.csr"
```

Durante la creación de este fichero .csr se nos pedirán una serie de datos, de los cuales deberá estar completo uno de ellos:

```
You are about to be asked to enter information that will be
incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished
Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:ES
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:abeleno@viewnext.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
```

Una vez tengamos el fichero .csr lo podremos firmar. Esto lo podremos hacer a través de una entidad certificadora o bien lo podremos hacer nosotros mismos.

En este caso, lo haremos nosotros mismos mediante el siguiente comando:

```
openssl x509 -signkey "service-principal.key" -in "service-
principal.csr" -req -days 365 -out "service-principal.crt"
```

Como podremos ver, lo hemos firmado proporcionándole una validez de un año (365 días), más que suficiente para el desarrollo de esta memoria.

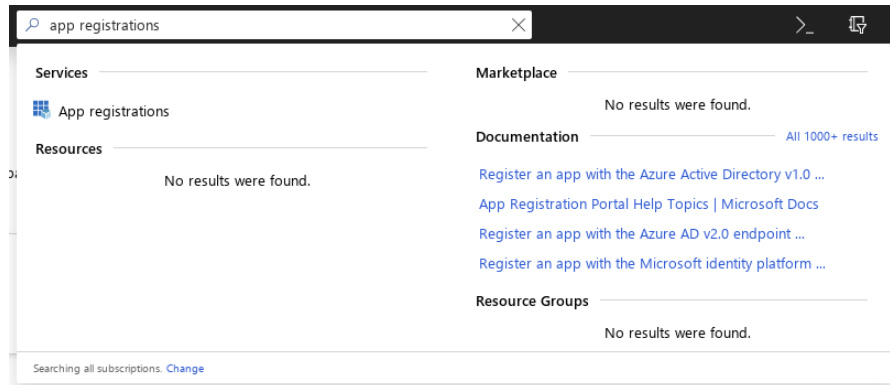
Nota: En el output del comando anterior se nos tiene que indicar que la firma se ha realizado correctamente, mediante "Signature ok".

Por último, tendremos que generar un fichero .pfx para poder autenticarnos en Azure:

```
openssl pkcs12 -export -out "service-principal.pfx" -inkey
"service-principal.key" -in "service-principal.crt"
```

Teniendo esto, a continuación crearemos el Service Principal. Esto lo haremos utilizando el propio portal de Azure, siguiendo los siguientes pasos:

Primero, iremos al apartado de búsqueda y buscaremos “App registrations”, y seleccionaremos el primer resultado:



Una vez en el menú de *App Registrations*, haremos click en el botón *Endpoints* y se nos abrirá un menú que contendrá una lista de URIs. De aquí, cogeremos la URI que nos dice *Oauth 2.0 authorization endpoint (v1)*, que contendrá un código con la siguiente estructura: `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxxxx`. Este código será nuestro *tenant_id*.

A continuación, cerraremos la ventana que contenía la lista de URIs y seleccionaremos *New registration*. Aquí, nos pedirá un nombre para nuestra aplicación. Podemos poner el nombre que queramos, en este caso pondremos “Proyecto Integrado”.

Register an application

* Name

The user-facing display name for this application (this can be changed later).

Supported account types

Who can use this application or access this API?

- Accounts in this organizational directory only (abeleno)
- Accounts in any organizational directory
- Accounts in any organizational directory and personal Microsoft accounts (e.g. Skype, Xbox, Outlook.com)

[Help me choose...](#)

Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web

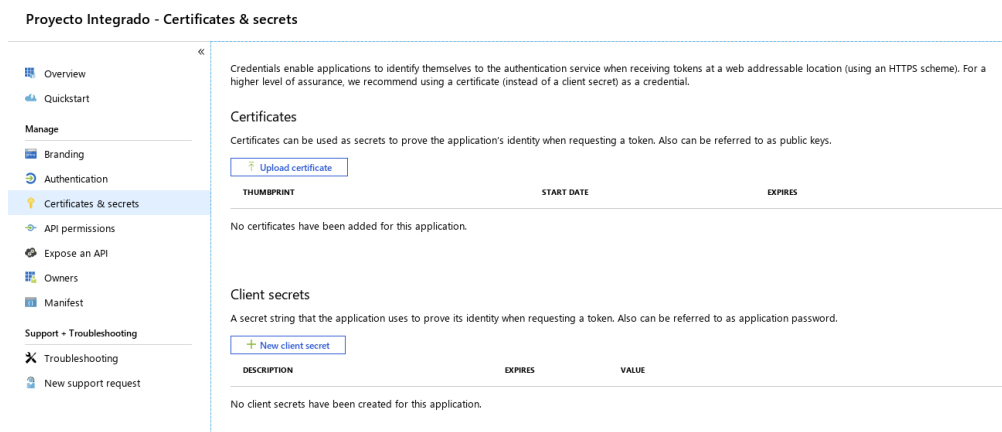
Nota: Como vemos en la captura anterior, podemos dejar en blanco el campo Redirect URI, ya que no disponemos de una.

Ahora, haremos click en *Register*. Ya tendremos creada nuestra aplicación en el Azure Active Directory.

Cuando la tengamos, se nos abrirá una ventana que contendrá varios datos de dicha aplicación entre los cuales estará *Application ID*. Guardaremos este GUID, ya que será el valor del campo *client_id* que necesitaremos para autenticarnos.

Lo siguiente que haremos será subir el fichero *.crt* que hemos generado anteriormente a Azure para así asociar este certificado con la aplicación que hemos creado en los pasos anteriores.

En la misma página en la que estábamos antes, es decir, en la de nuestra aplicación, seleccionaremos *Certificates & secrets*, y se nos abrirá un menú similar al siguiente:



Aquí, seleccionaremos la opción *Upload certificate* y se nos dará a elegir el fichero que queremos subir. En nuestro caso, seleccionaremos el fichero *.crt* que hemos creado anteriormente (*service-principal.crt*).

Por último, el valor del campo *subscription_id* será el GUID de la suscripción que queramos utilizar. Este GUID lo podremos conseguir desde la sección *Subscriptions* del portal de Azure, seleccionando la suscripción que estemos usando y copiando el valor de *Subscription ID*.

También podremos obtenerlo desde la Azure CLI, ejecutando el comando que vimos en la sección “Autenticación mediante la Azure CLI”:

```
az account list -o table
```

El GUID de la suscripción a usar nos aparecerá en el campo *SubscriptionId*.

Cuando tengamos todos estos valores, podremos exportarlos como variables de entorno (forma recomendada, con los nombres `ARM_CLIENT_ID`, `ARM_SUBSCRIPTION_ID` y `ARM_TENANT_ID`) o podremos tener los valores en un fichero `backend.tf`, cuya estructura vemos a continuación:

```
provider "azurerm" {
  subscription_id = "<valor de subscription_id>"
  client_id       = "<valor de client_id>"
  client_certificate_path = "<Ruta al certificado>"
  client_certificate_password = "<Contraseña del certificado>"
  tenant_id      = "<valor de tenant_id>"
}
```

En determinados casos, nos resultaría mejor este último tipo de autenticación, pero durante el desarrollo de este proyecto usaremos la autenticación mediante la Azure CLI.

Autenticación en Azure para Ansible

También disponemos de varios tipos de autenticación en Azure para Ansible.

Autenticación mediante la Azure CLI

Este tipo de autenticación se hará de la misma manera en la que se hizo para Terraform en el apartado “Autenticación mediante la Azure CLI”.

Más adelante indicaremos el otro requisito para poder autenticarnos con Ansible mediante la Azure CLI.

Autenticación mediante un fichero de credenciales

Una de las otras maneras posibles para lograr la autenticación en Azure será creando el fichero `~/.azure/credentials`, que contendrá los datos que Ansible necesitará para llevar a cabo la autenticación.

Lo que tendremos que hacer para conseguir dichos datos será crear otro *Service principal*, esta vez para manejar la autenticación con Ansible.

En esta ocasión, crearemos dicho *Service principal* mediante la Azure CLI usando el siguiente comando:

```
az ad sp create-for-rbac --name <nombre del Service Principal>
```

Al ejecutar este comando, el *output* que obtendremos será en formato JSON, que nos proporcionará los siguientes datos:

```
alvaro@arca:~$ az ad sp create-for-rbac --name ansible
Changing "ansible" to a valid URI of "http://ansible", which is
the required format used for service principal names
Retrying role assignment creation: 1/36
Retrying role assignment creation: 2/36
{
  "appId": "<valor de appId>",
  "displayName": "ansible",
  "name": "http://ansible",
  "password": "<valor de password>",
  "tenant": "<valor de tenant>"
}
```

Aquí, *appId* será nuestro *client_id*, *password* será nuestro *client_secret* y *tenant* será *tenant* en el fichero `~/azure/credentials` mencionado anteriormente.

Con todo esto, mas el GUID de la suscripción que queremos usar, ya tenemos todo crear el fichero *credentials*, que tendrá la siguiente estructura:

```
[default]
subscription_id=<GUID de la suscripción>
client_id=<valor de appId>
secret=<valor de password>
tenant=<valor de tenant>
```

Autenticación mediante variables de entorno

Para configurar la autenticación en Azure mediante variables de entorno, tendremos que ejecutar el mismo comando que en el apartado anterior.

Una vez tengamos el *output*, las variables de entorno serán las siguientes:

- *AZURE_SUBSCRIPTION_ID*: Que será el ID de la suscripción que usemos.
- *AZURE_CLIENT_ID*: Será el valor de *appId*.
- *AZURE_CLIENT_SECRET*: Será el valor de *password*.
- *AZURE_TENANT_ID*: Será el valor de *tenant*.

Y finalmente ya tendremos configurada la autenticación para Ansible. Como hemos dicho anteriormente para Terraform, trabajaremos con la autenticación mediante la Azure CLI para Ansible durante el desarrollo de esta memoria.

Autenticación en Azure para InSpec

De manera similar al resto de herramientas utilizadas hasta ahora, InSpec también nos permite autenticarnos en Azure mediante diferentes métodos.

Autenticación mediante fichero de credenciales

Una de las formas que tenemos para conseguir que InSpec se pueda autenticar en Azure será creando un fichero de credenciales como ya hicimos en el apartado “Autenticación mediante un fichero de credenciales” para Ansible.

Serán los mismos pasos, es decir, crearemos un *Service principal* con el siguiente comando:

```
az ad sp create-for-rbac --name inspec
```

En la salida de este comando, como ya hemos visto previamente, habrá una serie de datos. A nosotros nos interesa de nuevo *appId*, que será *client_id* en nuestro fichero *credentials*, *password* que será *client_secret*, y *tenant* que será *tenant_id*.

Por último, crearemos el fichero `~/.azure/credentials`, aunque esta vez con una estructura distinta al apartado mencionado anteriormente. El fichero debería quedar como sigue:

```
[ID de la suscripción]
client_id = "<valor de appId>"
client_secret = "<valor de password>"
tenant_id = "<valor de tenant>"
```

Autenticación mediante variables de entorno

Utilizando este método de autenticación, tendremos que crear un nuevo *Service principal* para esta aplicación, lo que haremos con el mismo comando utilizado en el apartado anterior.

Volveremos a utilizar el *output* de dicho comando para la autenticación, aunque en este caso no tendremos los valores en el fichero `~/.azure/credentials`, sino que definiremos variables de entorno a partir de ellos.

Las variables de entorno serían las mismas que utilizamos el apartado “Autenticación mediante variables de entorno” para Ansible.

Para InSpec, utilizaremos la autenticación mediante fichero de credenciales.

Anotaciones sobre los métodos de autenticación

Debemos anotar que si por ejemplo elegimos autenticar Ansible mediante el fichero de credenciales, no podremos elegir lo mismo para InSpec. Esto se debe a que dicho fichero de credenciales (`~/.azure/credentials`) tiene estructuras diferentes para ambas herramientas.

También tenemos que apuntar que, aparte de establecer la autenticación con Azure para InSpec, tendremos que asegurarnos de que el puerto 22 de las máquinas es accesible, ya que también realizaremos *tests* mediante SSH.

Pruebas de autenticación

Terraform

Con independencia del método que hayamos elegido para autenticarnos, ya deberíamos ser capaces de empezar a crear infraestructura en Azure con Terraform. Como prueba, vamos a crear una máquina virtual con el código que encontramos aquí: <https://gitlab.com/alvarobrod/proyecto-integrado/blob/master/autenticacion/terraform/main.tf>

Esto lo guardaremos en un fichero llamado *main.tf* y, desde el mismo directorio de dicho fichero, ejecutaremos el siguiente comando para inicializar la configuración de Terraform. Este comando lo debemos ejecutar primero siempre:

```
terraform init
```

Con el comando anterior, se nos debería descargar automáticamente el *plugin* para Azure.

A continuación, ejecutaremos el comando que elaborará el plan de ejecución que Terraform seguirá a la hora de crear y desplegar nuestra infraestructura:

```
terraform plan
```

El *output* del comando anterior, es decir, el plan de ejecución del que hablábamos anteriormente, será similar a lo siguiente:

```
(ansible) alvaro@arca:~/terraform$ terraform plan
[...]
Terraform will perform the following actions:

# azurerm_network_interface.mynic will be created
+ resource "azurerm_network_interface" "mynic" {
  + applied_dns_servers      = (known after apply)
  + dns_servers              = (known after apply)
  + enable_accelerated_networking = false
  + enable_ip_forwarding     = false
  + id                       = (known after apply)
  + internal_dns_name_label  = (known after apply)
  + internal_fqdn            = (known after apply)
  + location                 = "westeurope"
  + mac_address              = (known after apply)
  + name                     = "default-nic"
}
[...]
```

```
# azurerm_virtual_network.myvnet will be created
+ resource "azurerm_virtual_network" "myvnet" {
  + address_space = [
    + "10.0.0.0/16",
  ]
  + id              = (known after apply)
  + location        = "westeurope"
  + name            = "default-vnet"
  + resource_group_name = "myrsg"
  + tags            = (known after apply)

  + subnet {
    + address_prefix = (known after apply)
    + id              = (known after apply)
    + name            = (known after apply)
    + security_group = (known after apply)
  }
}
```

```
Plan: 5 to add, 0 to change, 0 to destroy.
[...]
```

Por último, ejecutaremos el comando que creará la infraestructura:

```
terraform apply
```

Este comando nos volverá a mostrar el plan de ejecución a seguir y nos pedirá que confirmemos la acción antes de proceder a crear los recursos:

```
alvaro@arca:~/terraform$ terraform apply
[...]
Terraform will perform the following actions:

# azurerm_network_interface.mynic will be created
+ resource "azurerm_network_interface" "mynic" {
  + applied_dns_servers = (known after apply)
  + dns_servers         = (known after apply)
  + enable_accelerated_networking = false
  + enable_ip_forwarding = false
  + id                  = (known after apply)
  + internal_dns_name_label = (known after apply)
  + internal_fqdn       = (known after apply)
  + location            = "westeurope"
  + mac_address         = (known after apply)
  + name                = "default-nic"
}
[...]
Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

Enter a value: yes

azurerm_resource_group.myrsg: Creating...
```

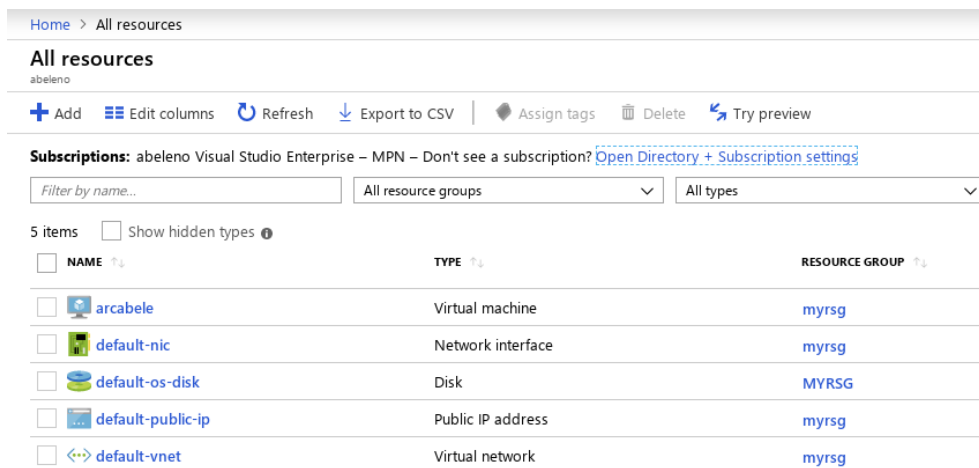


```
[...]
azurerms_virtual_machine.myvm: Creation complete after 2m34s
[id=/subscriptions/df068d66-bc96-4808-a7cd-fb273bab322c/resourceGroups/myrsg/providers/Microsoft.Compute/virtualMachines/arcabele]
```

Apply complete! Resources: 6 added, 0 changed, 0 destroyed.

Al comando anterior podremos añadirle `--auto-approve` para que no nos pida confirmación.

Tras esto, como podremos comprobar en el Portal de Azure, ya se habrá creado la máquina virtual:



Home > All resources

All resources
arcabele

+ Add Edit columns Refresh Export to CSV Assign tags Delete Try preview

Subscriptions: arcabele Visual Studio Enterprise – MPN – Don't see a subscription? [Open Directory + Subscription settings](#)

Filter by name... All resource groups All types

5 items Show hidden types

NAME	TYPE	RESOURCE GROUP
arcabele	Virtual machine	myrsg
default-nic	Network interface	myrsg
default-os-disk	Disk	MYRSG
default-public-ip	Public IP address	myrsg
default-vnet	Virtual network	myrsg

Ansible

Para comprobar si Ansible se puede autenticar correctamente en Azure, crearemos un *Resource group* de prueba. Para ello, tendremos un fichero *main.yml* que contendrá el siguiente código:

```
- name: Create Azure test Resource group
  hosts: localhost
  connection: local
  tasks:
    - name: Create resource group
      azure_rm_resourcegroup:
        name: rg-prueba
        location: westeurope
```

Una vez tengamos este fichero, ejecutaremos el siguiente comando:

```
ansible-playbook main.yml
```

Nota: Este comando, como se indicó en el apartado “Instalación mediante pip” lo podremos ejecutar siempre y cuando nuestro entorno virtual de Python esté activo, en el caso de que lo hayamos instalado de dicha manera.

La salida del comando anterior será similar a la siguiente:

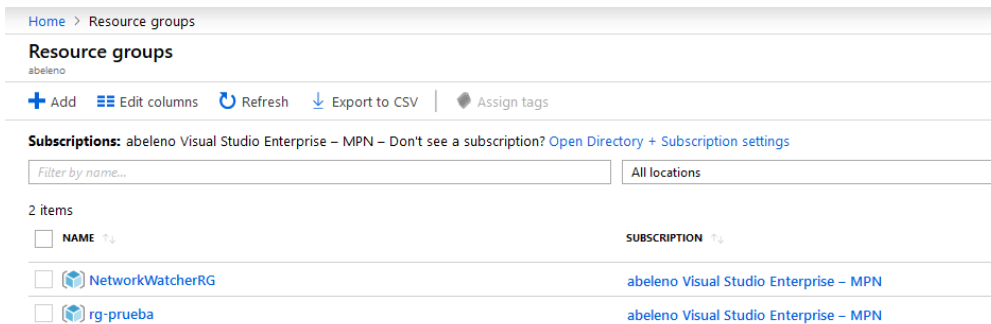
```
PLAY [Create Azure Resource group]
*****

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

TASK [Create resource group]
*****
changed: [localhost]

PLAY RECAP
*****
localhost: ok=2    changed=1    unreachable=0    failed=0
```

Acto seguido, si Ansible nos indica que ha ido todo bien y que no ha habido ningún fallo (Será así cuando en el apartado *PLAY RECAP* el valor de *failed* sea 0), podremos verificar mediante el Portal de Azure que efectivamente se ha creado nuestro *Resource group* de prueba de forma correcta:



NAME	SUBSCRIPTION
NetworkWatcherRG	abeleno Visual Studio Enterprise - MPN
rg-prueba	abeleno Visual Studio Enterprise - MPN

Tras hacer esta prueba, podremos proceder al borrado del *Resource group*.

InSpec

La prueba de autenticación que haremos con InSpec consistirá de un sólo comando, ya que la misma herramienta nos proporciona una manera de testear que la autenticación sea correcta. El comando es el siguiente:

```
inspec detect -t azure://
```

El *output* que obtendremos debería ser similar al siguiente:

```
alvaro@arca:~$ inspec detect -t azure://
```

```
Platform Details
```

```
Name:      azure
Families:  cloud, api
Release:   azure_mgmt_resources-v0.17.4
```

Uso de submódulos en el repositorio

Los submódulos de Git son una forma de incluir un repositorio dentro de nuestro repositorio como un directorio más. La manera de trabajar con estos submódulos es muy simple. Para añadir un submódulo, ejecutaremos el siguiente comando:

```
git submodule add <URL del repositorio> <Ruta>
```

Tras esto, ya tendremos el submódulo en nuestro repositorio “principal”, creándose así un fichero llamado *.gitmodules*, que contendrá la información sobre los submódulos que contenga nuestro repositorio.

El comando anterior ya clonará el contenido del repositorio indicado en la ruta que hayamos definido. Por el contrario, si volviéramos a clonar el repositorio de nuevo (O si lo clonara alguna otra persona), se debería usar el argumento *--recursive* a la hora de clonarlo para que se trajera consigo también los contenidos del repositorio que actúa como submódulo. Si no hiciéramos esto, tendríamos que entrar al directorio donde se encuentra dicho submódulo y ejecutar el siguiente comando:

```
git submodule update --init
```

Con esto, nos aseguramos de que nuestro fichero *.git/config* conoce que nuestro repositorio cuenta con un submódulo y clona el contenido del mismo en la ruta especificada anteriormente.

Al ser prácticamente igual a un directorio, trabajaremos con los submódulos de la misma manera que si tuviéramos el contenido de los mismos directamente en el repositorio principal.

Si se hicieran cambios en el repositorio submódulo, nos bastaría con hacer un *pull* para obtener dichos cambios.

Creación de la infraestructura (I)

Trabajando con módulos en Terraform

Hasta ahora, cuando hemos creado recursos con Terraform lo hemos hecho a partir de un fichero *main.tf* que contiene la definición de los propios recursos a crear.

A partir de este momento, trabajaremos con los módulos de Terraform.

Los módulos nos harán más fácil el proceso de creación de nuestra infraestructura, debido a que no tendremos que crear a mano todos los recursos necesarios para, por ejemplo, crear una máquina virtual, sino que con sólo un bloque de código que hará referencia a nuestro módulo de máquina virtual Linux crearemos dicho recurso.

En Terraform, estos módulos tienen la misma estructura que cualquiera de los ficheros `.yaml` que hemos visto y utilizado hasta ahora, con la diferencia de que en los módulos encontraremos que casi todos los valores están establecidos con variables, que se definen en un fichero aparte; `variables.tf`. Esto es una decisión personal, ya que encuentro que es más cómodo trabajar de esta manera cuando se trata de módulos, siendo así más simple y fácil cambiar el valor de cierto campo o tener directamente valores definidos por defecto sin que estén directamente presentes en el fichero `main.tf`.

Para hacer uso de los módulos, tendremos que disponer de un fichero `.tf` que llame al mismo. A continuación, veremos el contenido de nuestro `main.tf`, que llamará al módulo que tendremos en el repositorio del proyecto:

```
module "myvm" {
  source = "../modulos/vm1"
  address_space = "10.0.0.0/16"
  address_prefix = "10.0.10.0/24"
  virtual_machine_name = "arcabele"
}
```

Como podemos ver, hacemos referencia al módulo que tenemos en el directorio `modulos` desde `source`, donde le damos el valor `../modulos/vm1` para indicar que el origen del módulo a utilizar está en la ruta indicada. Las demás variables serán aquellas que no tengan ningún valor por defecto en el fichero `variables.tf` del módulo, y por lo tanto deberemos definir las en este fichero, como por ejemplo el nombre de nuestra máquina virtual (Definido mediante la variable `virtual_machine_name`).

Módulos cuyo origen es un repositorio Git

En lugar de tener el valor de `source` como una ruta en local, podríamos establecer una URI de un repositorio Git. Es decir, podríamos prescindir de tener los módulos en el mismo lugar que el fichero (O ficheros) que cree la infraestructura y pasar a tenerlos en remoto.

Para esto, el valor del campo `source` debería ser similar a lo siguiente:

```
source = "git::https://gitlab.com/alvarobrod/proyecto-integrado.git//terraform/modulos/vm1"
```

Tal y como vemos, tras la URI de Git hemos indicado la ruta en la que se encuentra nuestro módulo. Esto es necesario ya que, aunque sólo dispongamos de un módulo para este proyecto, Terraform no es capaz de saber en qué directorio exacto se encuentra dicho módulo necesario para la creación de la infraestructura, con lo cual tendremos que indicarlo nosotros.

Tanto habiendo puesto el valor de *source* en local como en remoto, seguiremos el mismo proceso para crear la infraestructura. Este proceso es el mismo que vimos en el apartado “*Terraform*”, es decir, ejecutar los siguientes comandos:

```
terraform init
```

Para inicializar la configuración de Terraform.

```
terraform plan
```

Para elaborar el plan de ejecución a seguir.

```
terraform apply [--auto-approve]
```

Para aplicar dicho plan de ejecución y finalmente crear los recursos deseados.

Trabajando con Dynamic Inventory de Ansible

¿Qué es el Dynamic Inventory de Ansible?

El *Dynamic Inventory* de Ansible es, como su propio nombre indica, un inventario dinámico.

Mientras que existe el típico inventario estático de Ansible, donde definimos en un fichero los equipos sobre los que se aplicarán los roles o tareas así como su dirección IP o FQDN (*Full Qualified Domain Name*), usando el *Dynamic Inventory* no nos hará falta tener un fichero *inventory* que defina los *hosts*, sino que Ansible (Ya sea mediante *plugins* o *scripts* de inventario) recupera la información sobre estos por nosotros, siendo así más sencillo trabajar en un entorno en el que, por ejemplo, las direcciones IP no son fijas.

¿Cómo se utiliza?

A partir de la versión 2.8.0 de Ansible (Lanzada el día 16 de mayo del 2019), se nos proporciona un plugin para el *Dynamic Inventory* de Ansible para Azure.

Este plugin requiere que tengamos un fichero de configuración, el cual deberá tener extensión *.yaml* o *.yml* y cuyo nombre deberá terminar en *azure_rm*. Como prueba, el fichero *auth_azure_rm.yml* tendrá el siguiente contenido:

```
plugin: azure_rm
include_vm_resource_groups:
- <nombre del Resource Group>
auth_source: cli

# En mi caso:

plugin: azure_rm
include_vm_resource_groups:
- myrsg
auth_source: cli
```

Si nos fijamos, el valor de *auth_source* es *cli*, eso significa que Ansible utilizará la sesión que tenemos establecida en la Azure CLI para autenticarse. Este es el último requisito que indicamos en el apartado “Autenticación mediante la Azure CLI” para Ansible.

Cuando tengamos este fichero, lo último que tendremos que hacer será instalar los módulos del SDK de Azure para Ansible. Esto lo haremos mediante *pip*, con el siguiente comando:

```
pip install ansible[azure]
```

Con todo esto, ya podremos hacer uso del inventario dinámico. Por ejemplo, podremos hacer *ping* a la máquina que hemos creado anteriormente:

```
ansible all -m ping -i auth_azure_rm.yml
```

Nota: Si nos sale un error diciéndonos que no se ha podido verificar la clave del host, añadiremos al fichero /etc/ansible/ansible.cfg lo siguiente:

```
host_key_checking = False
```

Si este fichero no existiera lo crearemos, dejándolo tal y como sigue:

```
[defaults]
host_key_checking = False
```

La salida del comando anterior debería ser similar a la siguiente:

```
(ansible) alvaro@arca:~/ansible$ ansible all -m ping -i
auth_azure_rm.yml
arcabele_daa6 | SUCCESS => {
  "changed": false,
  "ping": "pong"
}
```

Si es así, tendremos nuestro *Dynamic Inventory* correctamente configurado.

Nota 1: Al tiempo de escribir esta memoria, al ejecutar el comando anterior aparece un Warning indicando que el intérprete de Python podría cambiarse a otro diferente. Esto se debe a que, a partir de la versión 2.8, Ansible detecta el SO, la distribución y la versión del equipo y consulta en una tabla con los intérpretes de Python adecuados a las propiedades anteriormente detectadas. Debido a que en Debian 9 todavía se utiliza Python 2 por defecto, nos saltará este Warning, ya que debería ser Python3 (/usr/bin/python3). Esto podemos solucionarlo añadiendo la siguiente línea al fichero /etc/ansible/ansible.cfg, aunque tendremos que tener cuidado porque puede que no se puedan resolver algunas dependencias que haya en nuestra receta (En nuestro caso pasará esto, así que no añadiremos la línea al fichero):

```
interpreter_python = "/usr/bin/python3"
```

Nota 2: Podremos ver que detrás del nombre de la máquina virtual aparece una especie de identificador. Esto es propio de Ansible, es un hash que añade automáticamente para hacer única a la máquina virtual. Podemos deshabilitar esto añadiendo plain_host_names: True al fichero auth_azure_rm.yml.

Primeros tests con InSpec

Para empezar a trabajar con InSpec, tendremos que crear un perfil.

Como apuntan en la propia web de InSpec, los perfiles son “el núcleo de la experiencia del testing en InSpec”. Se utilizan para organizar los tests que deseemos hacer a nuestra infraestructura.

Crearemos un nuevo perfil con el siguiente comando:

```
inspec init profile <nombre del perfil>
# En mi caso:
inspec init profile ssh
```

El *output* del comando anterior debería ser similar al siguiente:

```
alvaro@PC00GJNQ:~/inspec$ inspec init profile ssh

===== InSpec Code Generator
=====

Creating new profile at /home/alvaro/inspec/ssh
• Creating file README.md
• Creating directory controls
• Creating file controls/example.rb
• Creating file inspec.yml
• Creating directory libraries
```

Este comando nos debería haber creado una estructura de ficheros y directorios como la siguiente:

```
├── ssh
│   ├── controls
│   │   └── example.rb
│   ├── inspec.yml
│   ├── libraries
│   └── README.md
```

Una vez tengamos el perfil creado correctamente, ya podremos empezar a desarrollar las pruebas que queramos hacer a nuestra infraestructura. Estas pruebas reciben el nombre de *controls* (De aquí en adelante *controles*), y los ficheros que los definen se situarán en el directorio *controls* dentro de nuestro perfil. Dichos controles tendrán extensión *.rb*.

Para empezar, tendremos que modificar el fichero *inspec.yml* para adaptarlo a nuestras necesidades. Quedará como sigue:

```
name: ssh
title: Perfil ssh de InSpec
maintainer: Álvaro Beleño
copyright: Álvaro Beleño
copyright_email: abeleno@viewnext.com
license: Apache-2.0
summary: Perfil ssh de InSpec para el Proyecto Integrado
version: 0.1.0
supports:
  - os-family: linux
```

Los valores de los campos *title*, *maintainer*, *copyright*, *copyright_email* y *summary* variarán dependiendo de quién escriba el perfil.

Por defecto, InSpec nos proporciona un control de ejemplo, con el siguiente contenido:

```
title 'sample section'

# you can also use plain tests
describe file('/tmp') do
  it { should be_directory }
end

# you add controls here
control 'tmp-1.0' do # A unique ID for this control
  impact 0.7 # The criticality, if this control fails.
  title 'Create /tmp directory' # A human-readable title
  desc 'An optional description...'
  describe file('/tmp') do # The actual test
    it { should be_directory }
  end
end
```


end

Como podemos ver, InSpec tiene una sintaxis bastante sencilla. Este control simplemente se asegura de que existe el directorio `/tmp`.

Si queremos ejecutar dicho test en nuestra máquina virtual, ejecutaremos el siguiente comando:

```
inspec exec ssh -t ssh://<usuario de la MV>@<IP de la MV>
```

Nota: Debido a que estamos ejecutando los test mediante SSH, hay que recordar que la máquina virtual a testear debe tener el puerto 22 abierto. También hay que apuntar que la IP pública de la máquina la obtendremos, de momento, del Portal de Azure.

La salida del comando anterior, si todo ha ido bien, debería ser similar a la siguiente:

```
alvaro@arca:~/inspec$ inspec exec ssh -t ssh://alvaro@<IP de la MV>
Profile: Perfil ssh de InSpec (ssh)
Version: 0.1.0
Target:  ssh://alvaro@<IP de la MV>
✓ tmp-1.0: Create /tmp directory
✓ File /tmp should be directory
File /tmp
✓ should be directory
Profile Summary: 1 successful control, 0 control failures, 0 controls skipped
Test Summary: 2 successful, 0 failures, 0 skipped
```

Aquí se nos indica que, efectivamente, nuestra máquina cuenta con el directorio `/tmp` y por lo tanto ha pasado los tests de manera satisfactoria.

Como hemos visto, tendremos que copiar la dirección IP pública de nuestra máquina virtual directamente desde el Portal de Azure. Esto puede llegar a ser poco práctico, ya que tenemos que consultar dicho Portal cada vez que vayamos a hacer una prueba sobre alguna de las máquinas de nuestra infraestructura.

Aquí es donde los *outputs* de Terraform nos son muy útiles.

InSpec y los outputs de Terraform

¿Qué son los outputs de Terraform?

Son una especie de variables que guardarán la información que indiquemos de nuestra infraestructura. Cuando tenemos *outputs* definidos, sus valores normalmente nos aparecen al finalizar la ejecución del *apply*.

¿Cómo se definen?

Estableceremos los *outputs* que deseemos en un fichero llamado *outputs.tf* (No es necesario que el fichero tenga ese nombre, pero es buena práctica que así sea. Igualmente, tampoco es necesario que definamos los *outputs* en un fichero aparte, pero de nuevo, es buena práctica).

Creamos un fichero *outputs.tf* con el siguiente contenido:

```
output "public_ip" {
    value = azurerm_public_ip.mypip.ip_address
}
```

El valor de *value* dependerá de la definición que hayamos hecho del recurso de la IP pública en el fichero *main.tf* de nuestro módulo. En mi caso, el recurso que crea la dirección IP pública utilizada por la máquina virtual se llama *mypip*.

Al estar trabajando con módulos, tendremos que crear otro *output*, ya que el que definimos para el módulo no nos aparecerá a menos que creemos un nuevo *output* cuyo valor sea la variable de salida del módulo.

Para esto, en el mismo directorio donde tenemos el fichero *main.tf* desde el que hemos llamado al módulo, crearemos un fichero *outputs.tf* con el siguiente contenido:

```
output "public_ip" {
    value = module.myvm.public_ip
}
```

De nuevo hay que apuntar que el valor de *value* dependerá de cómo hayamos llamado en este caso al módulo que hemos definido en el fichero *main.tf*. En este caso, lo llamamos *myvm*.

Una vez tengamos ambos ficheros listos, podremos ejecutar el siguiente comando, que actualizará el fichero de estado de Terraform con metadatos acordes a la infraestructura sobre la que estamos trabajando actualizándose también los *outputs*:

```
terraform refresh
```

La salida del comando anterior será parecida a la siguiente:

```
(ansible) alvaro@arca:~/terraform$ terraform refresh
module.myvm.azure_rm_resource_group.myrg: Refreshing state...
[id=/subscriptions/<ID de suscripción>/resourceGroups/myrg]
[...]
```

Outputs:

```
public_ip = <IP de la MV>
```

Con esto, ya tendremos la IP de nuestra máquina virtual sin necesidad de acceder al Portal de Azure.

Ahora, al ejecutar el siguiente comando podremos ver directamente el valor de la variable:

```
terraform output
```

La salida del comando anterior será similar a la siguiente:

```
(ansible) alvaro@arca:~/terraform$ terraform output
public_ip = <IP de la MV>
```

Sabiendo esto, podremos tener de manera muy sencilla dicho valor disponible como variable de entorno. Esto nos facilitará el proceso de trabajar con InSpec más adelante, ya que no tendremos que estar copiando y pegando esta dirección IP.

Accediendo a la IP desde la Azure CLI

También podremos obtener la dirección IP de nuestra máquina virtual (O máquinas virtuales) desde la Azure CLI.

Para ello, ejecutaremos el siguiente comando:

```
az vm list -o table -d
```

El *output* de este comando será una tabla donde se nos mostrará información sobre nuestra máquina virtual, como por ejemplo su nombre, su *Resource group* o su dirección IP:

```
alvaro@arca:~$ az vm list -o table -d
Name      ResourceGroup  PowerState  PublicIps  Fqdns
Location  Zones
-----  -
arcabele  MYRSG          VM running  <IP de la MV>
westeurope
```

Así, tendremos también otro método sencillo de acceder a la dirección IP de nuestra máquina virtual.

De “terraform output” a variables de entorno

Para pasar la salida del comando *terraform output* a una variable de entorno, primero tendremos que obtener sólo la dirección IP, es decir, sin “public_ip = ” delante. Esto lo haremos con este simple comando de *sed*:

```
terraform output | sed 's/public_ip = //'
```

Este comando ya nos proporcionará la dirección IP de nuestra máquina, con lo que podremos exportarla como una variable de entorno.

Si queremos agilizar aún más el proceso, podremos exportar como variable de entorno la salida del comando anterior, siendo de la siguiente forma:

```
export IP_ARCABELE=`terraform output | sed 's/public_ip = //'`
```

Shift left testing

El *shift left testing* es un enfoque sobre el *software* y los sistemas en el que el *testing* se empieza a realizar muy pronto en su ciclo de vida. Siguiendo este enfoque, se reducen el número de fallos que pueda haber y el número de acciones que tengamos que realizar para corregirlos durante dicho ciclo, lo cual ahorrará costes.

Creación de la infraestructura (II)

Para este proyecto, crearemos dos máquinas virtuales; Una será un servidor de base de datos y el otro será un servidor web que tendrá un gestor de contenidos (Drupal). Esto se desplegará mediante una receta de Ansible que es una modificación de la receta escrita para la asignatura ASO de 2º de ASIR, la cual se puede encontrar en el siguiente repositorio de GitHub: <https://github.com/alvarobrod/despliegue-ansible> (Para ver las modificaciones, iremos al directorio *ansible* del repositorio Git del proyecto, <https://gitlab.com/alvarobrod/proyecto-integrado>).

Terraform

Antes de crear dichas máquinas virtuales, borraremos los recursos ya existentes. Esto lo podemos hacer mediante el propio Portal de Azure o bien desde la línea de comandos con Terraform, al ejecutar este comando:

```
terraform destroy
```

Nota: Al anterior comando también podremos añadirle la opción --auto-approve como hemos hecho anteriormente si estamos seguros de que queremos borrar toda la infraestructura levantada con Terraform y no queremos que nos pida confirmación.

Una vez destruidos los recursos anteriores, tendremos que crear nuevos recursos. Para ello, utilizaremos dos módulos. El primero ya lo hemos utilizado anteriormente y crea todos los recursos necesarios para una máquina virtual. El segundo no lo hemos utilizado hasta ahora y sólo crea el recurso de la IP pública, el de la interfaz de red y el de la máquina virtual.

Al incluir este segundo módulo hemos definido también un segundo *output* en el primero, aparte de la dirección IP pública de la máquina virtual, y es la ID de la subred.

Esto será necesario para la creación de la máquina con el módulo VM2, ya que no creamos ninguna red ni subred, por lo tanto tendremos que pasarle el identificador de la subred creada con el módulo VM1.

Esta vez, el fichero *main.tf* que utilizaremos para crear la infraestructura será el siguiente:

```
module "myvm" {
  source = "../modulos/vm1"
  address_space = "10.0.0.0/16"
  address_prefix = "10.0.10.0/24"
  virtual_machine_name = "nodo1"
}

module "myvm2" {
  source = "../modulos/vm2"
  virtual_machine_name = "nodo2"
  subnet_id = module.myvm.subnet_id
}
```

Por otra parte, este será el contenido de nuestro fichero *outputs.tf*:

```
output "public_ip_n1" {
  value = module.myvm.public_ip
}

output "public_ip_n2" {
  value = module.myvm2.public_ip
}
```

Como podemos ver, en este fichero *outputs.tf* sólo tendremos las dos direcciones IP públicas de nuestras máquinas virtuales.

Cuando tengamos ambos ficheros, podremos ejecutar la sucesión de comandos ya vista anteriormente:

```
terraform init
terraform plan
terraform apply [--auto-approve]
```

El *output* de estos comandos será similar al que hemos visto anteriormente, aunque hay que apuntar que nos aparecerán los dos *outputs* que definimos anteriormente pero no nos aparecerá el valor de las direcciones IP. Esto es debido a que en Azure, las direcciones IP se asignan a las máquinas virtuales después de haber sido creadas, con lo cual Terraform (Como al terminar de crearse las máquinas virtuales termina la ejecución del comando, siendo dichas máquinas virtuales lo último que crea) no dispone de la información necesaria como para mostrarnos dichos *outputs*.

Para poder tener el valor de estas variables de salida tendremos que ejecutar, como vimos anteriormente, el siguiente comando:

```
terraform refresh
```

Tras hacer esto, podremos comprobar estas máquinas son accesibles a través de SSH.

Una vez creada la infraestructura, viene el turno de desplegar nuestra receta de Ansible para cada máquina virtual.

InSpec (Shift left testing)

Siguiendo el enfoque presentado anteriormente, ejecutaremos el primer perfil de InSpec para asegurarnos de que nuestra infraestructura está correctamente creada.

Iremos al directorio *inspec*, donde encontraremos el perfil *azure*, que contendrá el test que haremos a nuestra infraestructura. Y dentro del directorio *controls*, como ya indicamos anteriormente, tendremos los ficheros que correspondan a los controles que queramos realizar. En el perfil *azure*, estos controles están definidos en el fichero *virtual_machine.rb*, que tiene el siguiente contenido:

```
title 'Tests sobre Azure'
  desc 'La máquina virtual nodo1 debe existir'
  describe azurerm_virtual_machine(resource_group: 'myrsg',
name: 'nodo1') do
    it { should exist }
    its('name') { should eq('nodo1') }
    its('location') { should eq('westeurope') }
    its('type') { should
eq('Microsoft.Compute/virtualMachines') }
  end
  desc 'La máquina virtual nodo2 debe existir'
  describe azurerm_virtual_machine(resource_group: 'myrsg',
name: 'nodo2') do
    it { should exist }
    its('name') { should eq('nodo2') }
    its('location') { should eq('westeurope') }
    its('type') { should
eq('Microsoft.Compute/virtualMachines') }
  end
end
```

Para ejecutar este test, introduciremos el siguiente comando:

```
inspec exec azure -t azure://
```

La salida del comando anterior, si todo va bien, será similar a lo siguiente:

```
alvaro@arca:~/proyecto-integrado/inspec$ inspec exec azure -t
azure://
```

```
Profile: Perfil azure de InSpec (azure)
Version: 0.1.0
Target:  azure://df068d66-bc96-4808-a7cd-fb273bab322c
```

```
✓ azurerm_virtual_machine: Tests sobre Azure
  ✓ 'nodo1' Virtual Machine should exist
  ✓ 'nodo1' Virtual Machine name should eq "nodo1"
  ✓ 'nodo1' Virtual Machine location should eq
"westeurope"
  ✓ 'nodo1' Virtual Machine type should eq
"Microsoft.Compute/virtualMachines"
  ✓ 'nodo2' Virtual Machine should exist
  ✓ 'nodo2' Virtual Machine name should eq "nodo2"
  ✓ 'nodo2' Virtual Machine location should eq
"westeurope"
  ✓ 'nodo2' Virtual Machine type should eq
"Microsoft.Compute/virtualMachines"
```

```
Profile: Azure Resource Pack (inspec-azure)
Version: 1.2.0
Target:  azure://df068d66-bc96-4808-a7cd-fb273bab322c
```

```
No tests executed.
```

```
Profile Summary: 1 successful control, 0 control failures, 0
controls skipped
Test Summary: 8 successful, 0 failures, 0 skipped
```

Ansible

Primero, tenemos que asegurarnos de que nuestras máquinas son accesibles para Ansible. Siguiendo los pasos definidos en el apartado “¿Cómo se utiliza?”, haremos *ping* a los servidores mediante el siguiente comando:

```
ansible all -m ping -i auth_azure_rm.yml
```

Cuando hagamos esto, podremos empezar a aprovisionar nuestras máquinas.

Como hemos indicado anteriormente, tenemos una receta o *playbook* de Ansible para cada máquina virtual (No es posible hacerlo todo desde un sólo *playbook* como en el repositorio original, ya que el inventario es dinámico), y al mostrar nuestro inventario con el siguiente comando nos aparecen las dos máquinas, con lo que si ejecutáramos el *playbook* para el *nodo1* se aprovisionarían los dos servidores, tanto *nodo1* como *nodo2*:

```
ansible-inventory -i auth_azure_rm.py --graph
```

La salida del comando anterior nos mostrará las máquinas de nuestro inventario:

```
(ansible) alvaro@arca:~/ansible$ ansible-inventory -i
auth_azure_rm.yml --graph
@all:
  |--@ungrouped:
  |  |--nodo1_d2fb
  |  |--nodo2_34d7
```

Como es evidente, queremos que nos aparezca sólo la máquina que vamos a aprovisionar, para que no se produzcan conflictos. Esto lo podemos hacer de la siguiente manera.

Filtrado por nombre de las máquinas

Volvamos al fichero *auth_azure_rm.yml* que definimos en “¿Cómo se utiliza?”.

Sólo indicamos las opciones *plugin* (Obligatoria), *include_vm_resource_groups* y *auth_source*, pero podremos incluir más, como por ejemplo *exclude_host_filters*, que como su propio nombre indica, son filtros que nos ayudarán a excluir de nuestro inventario las máquinas que cumplan ciertos requisitos. Sabiendo esto, incluiremos lo siguiente al ya mencionado fichero *auth_azure_rm.yml*:

```
exclude_host_filters:
- name == 'nodo2'
```

Esto excluirá al *nodo2* de nuestro inventario, con lo cual sólo nos aparecerá *nodo1*:

```
(ansible) alvaro@arca:~/ansible$ ansible-inventory -i
auth_azure_rm.yml --graph
@all:
  |--@ungrouped:
  |  |--nodo1_d2fb
```

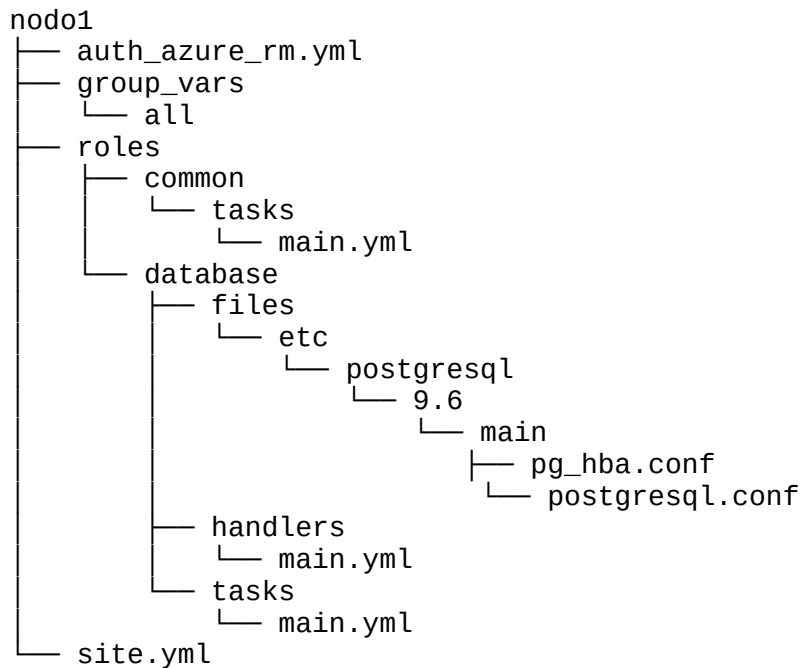
Y ya podremos proceder a la ejecución del *playbook*.

Cuando queramos excluir al *nodo1*, sólo tendremos que cambiar la última línea del fichero para que excluya a los *hosts* cuyo nombre sea *nodo1*.

Para más comodidad, en cada *playbook* de cada nodo tendremos un fichero *auth_azure_rm.yml* que excluirá al otro nodo.

Ejecutando los playbooks

La estructura del *playbook* del *nodo1* será la siguiente:



El contenido *site.yml* será el siguiente:

```

---
- name: Install PostgreSQL
  hosts: all
  become: yes
  roles:
    - common
    - database

```

Este fichero será el propio *playbook*, en el que indicaremos qué roles se ejecutarán (*common* y *database*).

Para ejecutar el *playbook* del *nodo1*, ejecutaremos el siguiente comando desde el directorio del mismo *playbook*:

```
ansible-playbook -i auth_azure_rm.yml site.yml
```

El *output* será parecido al siguiente:

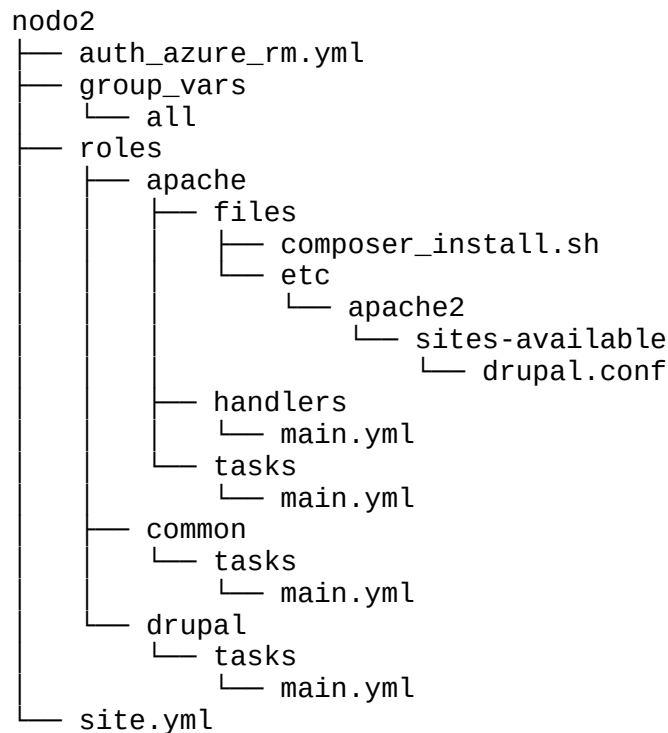
```

(ansible) alvaro@arca:~/ansible/nodo1$ ansible-playbook -i
auth_azure_rm.yml site.yml
[...]
PLAY RECAP
*****
nodo1: ok=8    changed=6    unreachable=0    failed=0
skipped=0    rescued=0    ignored=0

```

Ahora ejecutaremos el mismo comando, esta vez desde el directorio del *playbook* del *nodo2*.

La estructura del *playbook* del *nodo2* es la siguiente:



El contenido del fichero *site.yml* será el siguiente:

```

---
- name: Install Drupal with Apache and mod-php
  hosts: all
  become: yes
  roles:
    - common
    - apache
    - drupal

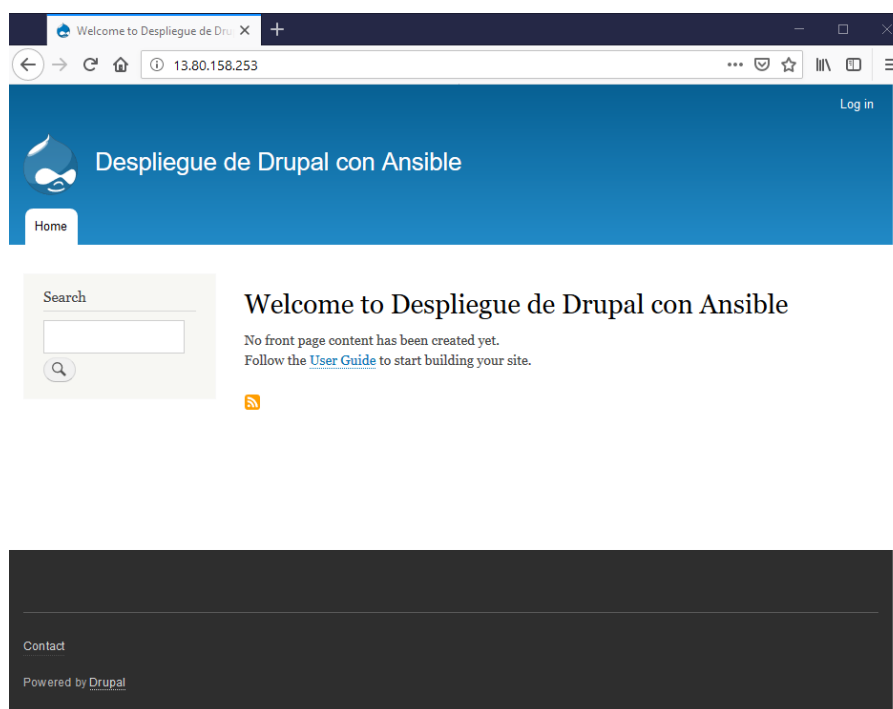
```

Aquí podremos ver que se ejecutarán los roles *common*, *apache* y *drupal*.

De nuevo, para ejecutar este *playbook* usaremos el siguiente comando:

```
ansible-playbook -i auth_azure_rm.yml site.yml
```

Cuando este termine de ejecutarse, si no ha habido ningún problema, podremos acceder al sitio que acabamos de crear mediante la IP del segundo servidor.



Y como podremos ver, nuestro sistema gestor de contenidos (En este caso Drupal) está instalado y es completamente funcional.

Ahora, ejecutaremos tests sobre nuestra infraestructura.

InSpec

Generando las variables de entorno

Como hicimos en el apartado “De “*terraform output*” a *variables de entorno*”, pasaremos las direcciones IP de nuestras máquinas virtuales a variables de entorno. Para ello, utilizaremos el mismo método que anteriormente pero añadiendo un *grep* de por medio. Veamos la salida del comando *terraform output*:

```
alvaro@arca:~/terraform/infraestructura2$ terraform output
public_ip_n1 = <IP de la MV1>
public_ip_n2 = <IP de la MV2>
```

Aquí podemos ver que el comando que utilizamos antes no valdría, ya que con este comando obtenemos dos direcciones IP, por lo tanto tendríamos que quitar una para poder guardar la otra como variable de entorno y viceversa. Como ya hemos dicho, esto lo haremos añadiendo un simple *grep*:

```
terraform output | grep n1 | sed 's/public_ip_n1 = //'
```

Este *grep* lo que hará será quedarse sólo con la línea en la que nos aparece *n1*, es decir, la IP del *nodo1*. Podríamos exportar dicha dirección IP de la misma manera que hicimos en el apartado antes mencionado, cambiando el comando cuya salida exportaremos por el de arriba.

Cuando queramos exportar la IP del *nodo2*, bastará con cambiar *n1* por *n2*.

Ejecutando los tests

Una vez hecho esto iremos al directorio *inspec*, en el que tendremos tres perfiles: *azure* (Que hemos utilizado anteriormente en el apartado “*Inspec (Shift left testing)*”), *nodo1* y *nodo2*. En el directorio *controls* del perfil *nodo1* tendremos los siguientes ficheros:

nodo1.rb, que contendrá los tests que aplicaremos al *nodo1*:

```
control 'Nodo 1' do
  title 'Tests del nodo1'
  desc 'El puerto 5432 debe ser alcanzable'
  describe host(ENV['IP_NOD01'], port: 5432, protocol: 'tcp')
  do
    it { should be_reachable }
  end
  desc 'telnet no debe estar instalado'
  describe package('telnet') do
    it { should_not be_installed }
  end
end
```

postgres.rb, que contendrá los tests que aplicaremos a PostgreSQL:

```
control 'Postgres' do
  title 'Tests de PostgreSQL'
  desc 'PostgreSQL debe estar instalado'
  describe package('postgresql-9.6') do
    it { should be_installed }
    its('version') { should cmp >= '9.6' }
  end
  desc 'PostgreSQL debe estar activado y funcionando'
  describe service('postgresql') do
    it { should be_enabled }
  end
end
```

Finalmente, en el perfil *nodo2* tendremos los siguientes ficheros:

apache.rb, que contendrá los tests que aplicaremos a Apache:

```
control 'Apache' do
  title 'Tests de Apache'
  desc 'Apache debe estar activado y funcionando'
  describe service('apache2') do
    it { should be_enabled }
    it { should be_installed }
  end
  describe file('/var/www/html') do
    it { should_not be_owned_by('www-data') }
  end
  describe http(ENV['IP_NOD02']) do
    its('status') { should eq 200 }
    its('headers.Content-type') { should include 'text/html' }
    its('body') { should include 'Despliegue' }
  end
end
```

nodo2.rb, que contendrá tests que aplicaremos al *nodo2*:

```
control 'Nodo 2' do
  title 'Tests del nodo2'
  desc 'El puerto 80 debe ser alcanzable'
  describe host(ENV['IP_NOD02'], port: 80, protocol: 'tcp') do
    it { should be_reachable }
  end
  desc 'telnet no debe estar instalado'
  describe package('telnet') do
    it { should_not be_installed }
  end
end
```

Para ejecutar estos tests, ejecutaríamos el perfil que corresponda con cada nodo de la siguiente forma, tal y como vimos en el apartado mencionado anteriormente:

```
# Para nodo1:
inspec exec nodo1 -t ssh://alvaro@$IP_NOD01
# Para nodo2:
inspec exec nodo2 -t ssh://alvaro@$IP_NOD02
```

Nota: Podemos ver que hemos utilizado las variables de entorno generadas anteriormente tanto para la ejecución de ambos comandos como en la definición de los propios controles.

Al ejecutar los comandos anteriores, si nuestra infraestructura y aplicaciones cumplen con los tests anteriormente establecidos, se nos mostrará un *output* parecido al siguiente:

```
alvaro@arca:~/inspec$ inspec exec nodo1 -t
ssh://alvaro@$IP_NODO1
[...]
```

```
✓ Nodo1: Tests del nodo1
  ✓ Host <IP del nodo1> port 5432 proto tcp should be
  reachable
  ✓ System Package telnet should not be installed
✓ Postgres: Tests de PostgreSQL
  ✓ System Package postgresql-9.6 should be installed
  ✓ System Package postgresql-9.6 version should cmp >=
  "9.6"
  ✓ Service postgresql should be enabled
```

```
Profile Summary: 2 successful controls, 0 control failures, 0
controls skipped
Test Summary: 5 successful, 0 failures, 0 skipped
```

Dependiendo del perfil que ejecutemos y los tests que contenga, la salida de este comando variará.

Otros apuntes

Durante el desarrollo de esta memoria, para el formato del código Terraform hemos utilizado el siguiente comando:

```
terraform fmt
```

Ejecutando ese comando en un directorio que contenga ficheros con extensión *.tf*, les dará a todos estos el mismo estilo y formato.

Alternativas

Terraform

Tal y como podemos hacer con Terraform, Ansible también nos da la opción de levantar infraestructura, incluso podremos hacer un “*Dry run*”, que básicamente es que el *playbook* se ejecutará pero no hará ningún cambio en el sistema/plataforma destino. En vez de eso, Ansible reportará los cambios que se habrían hecho en dicho sistema/plataforma. Esto es similar al *terraform plan*, sólo que en Ansible, la instrucción *Dry run* nos proporciona menos cantidad de información.

También, como alternativa, tendremos una herramienta *open source* llamada Pulumi. Con Pulumi, no tendremos que escribir el código que define la infraestructura en código propietario, sino que podremos escribirlo en lenguajes como Python, Go o JavaScript. Esta herramienta se puede integrar con Terraform, adaptando sus *plugins* de proveedores a Pulumi.

También podemos mencionar a AWS CloudFormation, aunque esta alternativa sólo nos sería útil en el caso de que estuviéramos trabajando con Amazon Web Services, ya que CloudFormation es una herramienta diseñada para trabajar sólo con dicho proveedor.

Azure

Para Azure, contamos con muchísimas alternativas, como por ejemplo OpenStack o AWS.

OpenStack es una opción ya conocida, al haber trabajado con ella en el centro durante el transcurso del ciclo formativo. Es una solución *cloud open source* para crear nubes tanto públicas como privadas. Cuenta con un gran número de componentes que proveen distintas funcionalidades, tales como *Heat*, que proporciona orquestación, *Horizon*, que proporciona el frontal gráfico, o *Swift*, que proporciona el almacenamiento de objetos. Esto hace que OpenStack nos de mucha flexibilidad a la hora de implementarlo, ya que podremos escoger los elementos que nos interesen en nuestra instalación.

AWS (Amazon Web Services) es la visión de Amazon. Es un proveedor *cloud* que, de manera similar a Azure, tiene un modelo de pago por consumo de recursos. Esta plataforma también ofrece productos relacionados por ejemplo con la robótica, como *AWS RoboMaker*, con el *IoT* (Internet de las cosas, o *Internet of Things* en inglés), como *AWS IoT Core* o con los servicios multimedia, como *AWS Elemental MediaTailor*.

Ansible

Como alternativas a Ansible, podríamos presentar (Entre otras opciones) a SaltStack y Puppet.

SaltStack es una opción bastante parecida a Ansible en la manera de trabajar, aunque tiene ciertas diferencias. Por ejemplo, SaltStack trabaja con un *Master* que controla a uno o varios *Minions*. Es decir, será el *Master* el que comunique a los *Minions* las tareas a ejecutar. También introduce conceptos nuevos como los *grains*, que son piezas de información del sistema (O sistemas) que estemos manejando, como por ejemplo su dirección IP, sistema operativo, kernel, etc. En cambio, la sintaxis que utilizaremos para definir dichas tareas será YAML con posibilidad de poder integrar plantillas Jinja, de manera similar a Ansible.

Por otro lado, tenemos a Puppet. Esta herramienta trabaja con un enfoque parecido al de SaltStack: Tendremos un nodo *master* y uno o varios nodos *agent*. A diferencia de esta última herramienta y Ansible, Puppet no se escribe en YAML, sino que se escribe

en su propio lenguaje, mediante el cual estableceremos *manifests*, no *playbooks* como en Ansible.

InSpec

Una alternativa interesante a InSpec podría ser el *Secure DevOps Kit for Azure* (También llamado AzSK, como abreviación). Como su propio nombre indica, es un kit de seguridad para Azure que se compone de diversos *scripts*, herramientas, extensiones, etc. Su finalidad es llegar a tener un *workflow* de DevOps seguro en las 6 áreas que se establecen, que abarcan desde la seguridad de nuestra suscripción a Azure hasta la integración de la seguridad en la Integración y despliegue continuos.

Cuando ejecutamos alguno de los tests, AzSK irá mostrándonos información sobre el mismo en la consola y, cuando termine dicho test, exportará los resultados a un fichero junto con información sobre los distintos controles que tenga el test ejecutado en concreto y recomendaciones sobre cómo solucionar los errores que haya podido haber.

La desventaja de esta herramienta es que no funciona en sistemas Linux, sólo podemos instalarla y manejarla desde sistemas Windows.

Siguientes pasos

Como siguientes pasos, podríamos considerar varias ideas:

Podríamos, por ejemplo, incluir un mayor número de tests a nuestros perfiles de InSpec. Según las necesidades que tengamos o, también, según determinadas pautas que se puedan marcar, podríamos hacer perfiles para controlar ciertas normas de seguridad y así saber en todo momento que el estado de nuestra infraestructura es bueno y que no sufre de ninguna vulnerabilidad contemplada en nuestros tests.

También sería buena idea automatizar el proceso de *testing* de nuestra infraestructura, incluyéndolo en un *pipeline* de CI (*Continuous Integration*, Integración Continua en inglés) que, por ejemplo, ejecute los tests cada vez que se produzca algún cambio en nuestra infraestructura y así tenerla controlada en todo momento.

Conclusiones

Como hemos podido ver durante el desarrollo de esta memoria, las tres herramientas utilizadas tienen un buen grado de integración tanto entre ellas como con la plataforma utilizada.

También, como hemos visto en el apartado anterior, esta combinación de herramientas nos puede traer muchas posibilidades, ya que manejar y configurar nuestra

infraestructura se nos hace muy sencillo, definiendo sólo unas líneas de código. Esto se ve facilitado por el uso de módulos y los Git *submodules* implementados en nuestro repositorio.

En definitiva, veo en estas herramientas una combinación perfecta para automatizar y controlar nuestra infraestructura de un modo cómodo y sencillo. Se puede adaptar a cualquier tamaño, siendo desde un despliegue simple como hemos hecho en este proyecto a despliegues con más envergadura y configuraciones más complicadas, pudiendo adaptar los tests con facilidad a nuestras necesidades.

Webgrafía

- [Documentación de Terraform](#)
- [Documentación de Microsoft Azure](#)
- [Documentación de Microsoft Azure \(II\)](#)
- [Documentación de Ansible](#)
- [Documentación de InSpec](#)
- [Documentación de Pulumi](#)
- [Documentación de OpenStack](#)
- [Documentación de AWS](#)
- [Documentación de SaltStack](#)
- [Documentación de Puppet](#)
- [Documentación de Secure DevOps Kit for Azure](#)
- [Artículo sobre Shift Left Testing y DevOps](#)
- [Página de Shift Left Testing en Wikipedia](#)