**Infraestructura como Código**

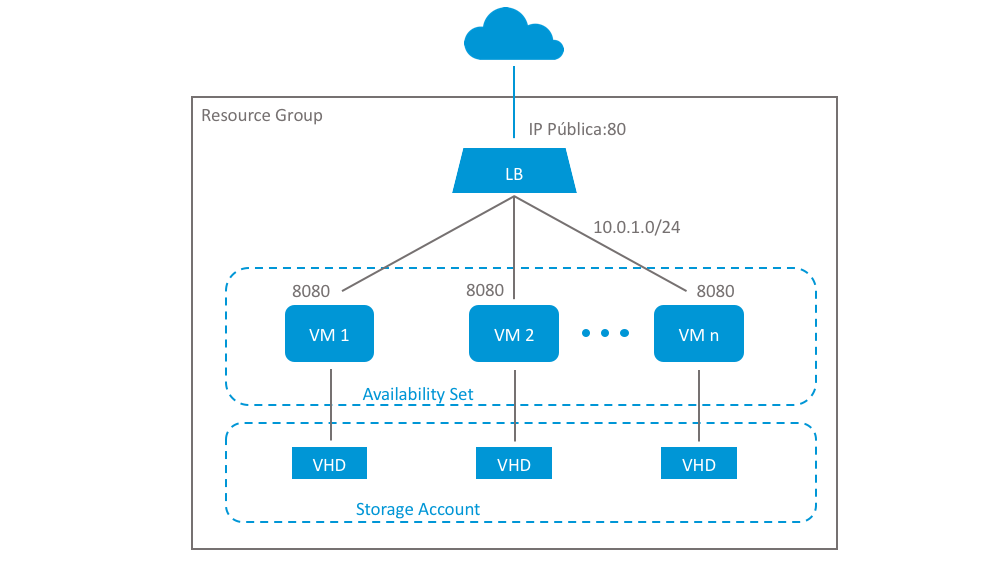
La **Infraestructura como Código** es un elemento clave en los equipos **Agile**.

En el artículo anterior vimos la importancia de orientar nuestra infraestructura a **producto** y cómo **Terraform** puede ayudarnos a agilizar su creación y mantenimiento de forma automatizada.

#### Nuestro objetivo

En este artículo vamos a ver un ejemplo práctico: describiremos infraestructura en **Azure** utilizando **Terraform**. Para ello, contaremos con el siguiente repositorio de GitHub donde está todo listo para que no tengas que escribir nada: https://github.com/drhelius/terraform-azure-demo

Este es nuestro escenario objetivo:



Crearemos un **clúster** de máquinas que expondrán una misma aplicación en el puerto 8080. Cada máquina tendrá su propio **disco virtual** dentro de la misma cuenta de almacenamiento.

Para garantizar la **alta disponibilidad** del servicio y la distribución uniforme de la carga utilizaremos un **balanceador** (LB).

Configuraremos el LB para que incluya de manera automática a todas las máquinas que conforman el cluster, es decir, las que se encuentran en el mismo **Availability Set** y cuyo servicio esté operativo. Si, por el contrario, su servicio está indisponible, se sacarán del grupo de balanceo.

Esto último lo conseguiremos poniendo una sonda, o **Probe**, que compruebe el puerto 8080 de cada máquina.

Por último, para poder acceder desde el exterior, asignaremos una **IP pública** al LB, abriremos el puerto 80 y lo mapearemos con el puerto 8080 de las máquinas que forman parte del cluster.

#### Manos a la obra

Antes de empezar, tendremos que preparar nuestra cuenta de Azure para poder usarla con Terraform, para lo cual seguiremos la documentación oficial.

Después de este paso contaremos con las **credenciales** necesarias para enlazar Terraform con nuestra cuenta de Azure.

Empezaremos creando una carpeta para nuestro proyecto y dentro, el fichero **demo.tf**, donde vamos a describir todos los elementos de la infraestructura para este artículo.

Puedes ver el fichero completo aquí: https://github.com/drhelius/terraform-azure-demo/blob/master/demo.tf

Antes de crear el Resource Group debemos proporcionar los datos necesarios del **Provider de Azure**:

provider "azurerm" {

client\_id = "${var.azure\_client\_id}"

client\_secret = "${var.azure\_client\_secret}"

subscription\_id = "${var.azure\_subscription\_id}"

tenant\_id = "${var.azure\_tenant\_id}"

}

Una vez declarada la forma de "conectarnos" ya podemos declarar nuestro nuevo **Resource Group**:

resource "azurerm\_resource\_group" "demo" {

name = "demo-terraform"

location = "${var.azure\_location}"

}

Como puedes ver, la configuración del Provider la hacemos con variables. Las **variables** en Terraform tienen este aspecto:

${var.mi\_variable}

Estas variables se tienen que declarar en un fichero llamado **variables.tf**:

variable "azure\_client\_id" {

type = "string"

}

variable "azure\_client\_secret" {

type = "string"

}

variable "azure\_location" {

type = "string"

default = "West Europe"

}

variable "azure\_subscription\_id" {

type = "string"

}

variable "azure\_tenant\_id" {

type = "string"

}

variable "demo\_instances" {

type = "string"

default = "2"

}

variable "demo\_admin\_password" {

type = "string"

}

En la declaración podemos poner valores por defecto. El resto de valoremos podemos proporcionarlos de muchas maneras: por **línea de comandos**, como **variables de entorno**, usando **Vault** o utilizando un fichero especial llamado **terraform.tfvars**.

Este fichero es útil para variables **sensibles** que deseemos suministrar en **local**. Lo ubicamos en la raíz del proyecto y Terraform lo lee automáticamente cuando realizamos cualquier operación:

azure\_client\_id = "xxxxxx-xx-xx-xx-xxxxxxx"

azure\_tenant\_id = "xxxxxx-xx-xx-xx-xxxxxxx"

azure\_client\_secret = "xxxx"

azure\_subscription\_id = "xxxxxx-xx-xx-xx-xxxxxxx"

A continuación, declaramos los datos de la **red privada** para el clúster:

resource "azurerm\_virtual\_network" "demo" {

name = "demo-virtual-network"

address\_space = ["10.0.0.0/16"]

location = "${var.azure\_location}"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

}

resource "azurerm\_subnet" "demo" {

name = "demo-subnet"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

virtual\_network\_name = "${azurerm\_virtual\_network.demo.name}"

address\_prefix = "10.0.1.0/24"

}

Podemos observar que en la **subred** se hace referencia a la **red virtual** ya declarada. El orden de declaración en el fichero no es importante.

Seguimos con una **IP pública** para el LB y con una **interfaz de red** (NIC) que utilizaremos en cada VM:

resource "azurerm\_public\_ip" "demo" {

name = "demo-public-ip"

location = "${var.azure\_location}"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

public\_ip\_address\_allocation = "static"

}

resource "azurerm\_network\_interface" "demo" {

count = "${var.demo\_instances}"

name = "demo-interface-${count.index}"

location = "${var.azure\_location}"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

ip\_configuration {

name = "demo-ip-${count.index}"

subnet\_id = "${azurerm\_subnet.demo.id}"

private\_ip\_address\_allocation = "dynamic"

load\_balancer\_backend\_address\_pools\_ids = ["${azurerm\_lb\_backend\_address\_pool.demo.id}"]

}

}

Hemos utilizado un atributo llamado **"count"** para crear más de un elemento de tipo "azurerm\_network\_interface" y para ello hemos utilizado una varible (var.demo\_instances).

Seguimos con todos los elementos necesarios para crear el **balanceador** en Azure: LB, regla de balanceo, probe, addres pool y availability set. Todos ellos debidamente referenciados entre sí:

resource "azurerm\_lb" "demo" {

name = "demo-lb"

location = "${var.azure\_location}"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

frontend\_ip\_configuration {

name = "default"

public\_ip\_address\_id = "${azurerm\_public\_ip.demo.id}"

private\_ip\_address\_allocation = "dynamic"

}

}

resource "azurerm\_lb\_rule" "demo" {

name = "demo-lb-rule-80-8080"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

loadbalancer\_id = "${azurerm\_lb.demo.id}"

backend\_address\_pool\_id = "${azurerm\_lb\_backend\_address\_pool.demo.id}"

probe\_id = "${azurerm\_lb\_probe.demo.id}"

protocol = "tcp"

frontend\_port = 80

backend\_port = 8080

frontend\_ip\_configuration\_name = "default"

}

resource "azurerm\_lb\_probe" "demo" {

name = "demo-lb-probe-8080-up"

loadbalancer\_id = "${azurerm\_lb.demo.id}"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

protocol = "Http"

request\_path = "/"

port = 8080

}

resource "azurerm\_lb\_backend\_address\_pool" "demo" {

name = "demo-lb-pool"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

loadbalancer\_id = "${azurerm\_lb.demo.id}"

}

resource "azurerm\_availability\_set" "demo" {

name = "demo-availability-set"

location = "${var.azure\_location}"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

}

Lo que conseguimos es que el LB balancee las máquinas que utilizan IPs del address pool.

Seguimos con el **almacenamiento**, un contenedor por VM en la misma cuenta de storage:

resource "azurerm\_storage\_account" "demo" {

name = "demoterraformstorage"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

location = "${var.azure\_location}"

account\_type = "Standard\_LRS"

}

resource "azurerm\_storage\_container" "demo" {

count = "${var.demo\_instances}"

name = "demo-storage-container-${count.index}"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

storage\_account\_name = "${azurerm\_storage\_account.demo.name}"

container\_access\_type = "private"

}

Y finalmente la descripción de la **VM** en sí:

resource "azurerm\_virtual\_machine" "demo" {

count = "${var.demo\_instances}"

name = "demo-instance-${count.index}"

location = "${var.azure\_location}"

resource\_group\_name = "${azurerm\_resource\_group.demo.name}"

network\_interface\_ids = ["${element(azurerm\_network\_interface.demo.\*.id, count.index)}"]

vm\_size = "Standard\_A0"

availability\_set\_id = "${azurerm\_availability\_set.demo.id}"

storage\_image\_reference {

publisher = "Canonical"

offer = "UbuntuServer"

sku = "16.04-LTS"

version = "latest"

}

storage\_os\_disk {

name = "demo-disk-${count.index}"

vhd\_uri = "${azurerm\_storage\_account.demo.primary\_blob\_endpoint}${element(azurerm\_storage\_container.demo.\*.name, count.index)}/demo.vhd"

caching = "ReadWrite"

create\_option = "FromImage"

}

delete\_os\_disk\_on\_termination = true

delete\_data\_disks\_on\_termination = true

os\_profile {

computer\_name = "demo-instance-${count.index}"

admin\_username = "demo"

admin\_password = "${var.demo\_admin\_password}"

custom\_data = "${base64encode(file("${path.module}/provision.sh"))}"

}

os\_profile\_linux\_config {

disable\_password\_authentication = false

}

}

Con todo esto preparado y ubicados en la ruta donde se encuentre nuestro fichero **demo.tf**, ejecutamos este comando:

$ terraform init

Lo cual inicializa nuestro proyecto y se descarga los providers necesarios, en nuestro caso el de Azure. Después podemos ejecutar lo siguiente:

$ terraform plan

Este comando nos informa **sin hacer cambios** de lo que sucedería si aplicásemos la configuración que hemos descrito.

Nos solicitará por línea de comandos el valor de aquellas **variables que no hayamos establecido**. En nuestro caso, el password de las VMs:

var.demo\_admin\_password

Enter a value:

Después veremos una descripción de los recursos que se crearían o modificarían en Azure:

Plan: 16 to add, 0 to change, 0 to destroy.

Ya estamos listos para **aplicar** los cambios y que todo se cree de manera **automática** en Azure. Para ello, ejecutamos lo siguiente:

$ terraform apply

Poco a poco, los elementos empezarán a crearse. Una vez terminado nos informará del **resultado** y nos mostrará las **variables de salida** (outputs) que hayamos definido en el fichero **output.tf**, en nuestro caso, la IP pública del LB:

output "lb\_public\_ip" {

value = "${azurerm\_public\_ip.demo.ip\_address}"

}

Veríamos algo así:

Apply complete! Resources: 16 added, 0 changed, 0 destroyed.

The state of your infrastructure has been saved to the path

below. This state is required to modify and destroy your

infrastructure, so keep it safe. To inspect the complete state

use the `terraform show` command.

State path:

Outputs:

lb\_public\_ip = 52.233.156.135

En este momento podemos poner la **IP pública** en nuestro **navegador** y probarlo.

La aplicación de ejemplo es un servicio web dentro de un **contenedor Docker** que devuelve el **hostname** del contenedor donde se está ejecutando. Cada vez que actualicemos la página deberíamos ver hasta **2 hostnames diferentes**:

Hello World from host "d270d4254e38".

Ahora vamos a modificar el valor de la variable demo\_instances en el fichero **variables.tf** y lo aumentamos, por ejemplo, a **4**:

variable "demo\_instances" {

type = "string"

default = "4"

}

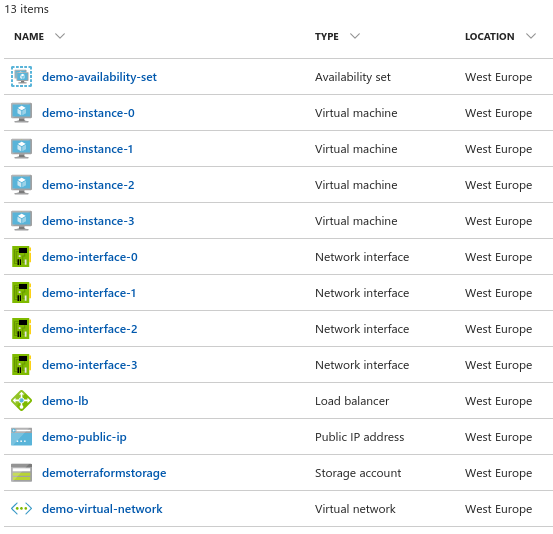
Volvemos a ejecutar el comando terraform apply y veremos cómo se crearán los elementos que **faltan**, el resto se queda como está:

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

Es decir, se crearán las dos **nuevas VMs** con sus interfaces de red, almacenamiento, etc. Además, se incluirán en el grupo de balanceo y la aplicación ahora estará disponible desde 4 servidores utilizando la **IP pública** del LB.

Podemos poner de nuevo la **IP pública** en nuestro **navegador** para probarlo. Cada vez que actualicemos la página ahora deberíamos ver hasta **4 hostnames diferentes**.

Si entramos en nuestra cuenta de Azure veríamos todos los elementos creados:



Esto es sólo el **comienzo** ya que con **Terraform** podemos modelar escenarios muchos más **complejos**, con centenares de elementos y todo ello versionado bajo **Git** o cualquier otro SCM.