Project: Coalfire Lab

## Author: Christian Maytin

## Prerequisites:

* Terraform
* Azure Tenant with Pay-As-You-Go Subscription

## Procedure

Initial setup proceeded with the creation of the Azure tenant and subscription. Take care during this part as it is important to take note of which accounts you are using to create the tenant and sub. Due to previous training/testing exercises in the past, I have run into conflicts and been subsequently unable to take advantage of the Azure Trial account offering by Microsoft.

First order of business after the Azure tenant is fully up, is to create a control resource group and VM from where I can setup terraform and the development environment. While the VM is provisioning, I also set up the Azure Cloudshell for the CLI.

No problems encountered and as a bonus, the Azure Cloudshell comes preinstalled with Terraform.

A good place to look is at the [Microsoft knowledge base on Terraform](https://docs.microsoft.com/en-us/azure/developer/terraform/), as it turns out, they happen to have a decent amount of documentation on Terraform. A few of the articles in this page have examples on some of the configuration necessary to complete this implementation.

Once the VM comes up, I download the Terraform executable from the Terraform download page. The executable is added to the environmental path variables as I am using a Windows 11 Enterprise VM to accomplish this. After testing entering terraform init in the new Windows Terminal and receiving an error from the Terraform runtime, I have confirmation that the installation is working as expected.

To more easily design and code in HCL, I download Visual Studio Code and also install the Terraform Extension and Terraform Azure Extension. I confirm that the IDE is working properly and intellisense is functional.

On Azure Cloudshell, I set up the Service Principal I will be using to authenticate against Azure from the Terraform executable. The following [documentation](https://docs.microsoft.com/en-us/azure/developer/terraform/authenticate-to-azure?tabs=bash) is very useful. Once the Service Principal has been configured, I set up an empty main.tf and the provider block with the newly created credentials. Terraform is able to authenticate successfully.

I make note that plain text credentials are not best practice and to redress this part of the implementation later if time allows.

At this point, I populate the terraform configuration with the skeleton of the architecture. All of the high level resource blocks are created, to later be filled out with the proper parameters and fields.

There are various sources I use during this step:

<https://docs.microsoft.com/en-us/azure/developer/terraform/>

<https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>

The Terraform registry is invaluable. I will reference the registry repeatedly throughout the entire configuration to determine the proper arguments/attributes for each resource. It is no exaggeration to say that implementation in this time frame would have likely been impossible without it.

After I finish setting up the initial resources and fill out their arguments, I delve deeper into the two aforementioned knowledge bases for how to best implement the requirements. A look at the following [article](https://docs.microsoft.com/en-us/azure/developer/terraform/create-vm-cluster-with-infrastructure) helps point me in the right direction and I realize that I will need to do a rewrite of my code for the high availability VMs.

Interpolation becomes important at this point for successful implementation of resources with multiple instances, the following [documentation](https://www.terraform.io/docs/configuration-0-11/interpolation.html) proves very helpful.

I complete the rewrite and test. A few errors require me to clean up the syntax and fill in missing arguments. Terraform finally gives me a clean bill of health and I can proceed further into the implementation.

At this point, I run terraform apply and terraform creates the resources. Terraform completes the command and I inspect the implementation. There are a few design errors that need remediation, namely, with the NSG rules. At this point, I’ve discovered that I cannot associate an NSG using the subnet block, as evidenced [here](https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/subnet). Instead, I have to use a resource “association” in order to accomplish this. I ultimately decide to attach the NSG at the NIC level, rather than the subnet layer, to be as granular as possible.

This resolves the issues I found during testing and I can continue implementing the remaining resources.

I can now turn my attention to the Azure Load Balancer and the Apache VM. Using the example in the [documentation](https://docs.microsoft.com/en-us/azure/developer/terraform/create-vm-cluster-with-infrastructure), I set up the resource blocks and populate the arguments, despite a nagging suspicion that the code is incomplete. My fears are, unfortunately, realized. The resources are created by Terraform without a problem, but the VM is inaccessible, and the ALB is simply not working.

A visit to the Terraform registry reveals many more resources associated with the ALB. Looking over the Microsoft [documentation](https://docs.microsoft.com/en-us/azure/load-balancer/quickstart-load-balancer-standard-internal-portal?tabs=option-1-create-internal-load-balancer-standard) on the load balancer also reveals the necessity of having a backend pool (which is precisely what I thought was missing on the previous example) and a load balancing rule. Setting up the remaining associations and resources, the ALB was now functional.

The issue now remained with the Apache VM. I needed a way to automate the installation of the Apache environment. I had explored a few different ways to do this earlier, via provider, via custom data, and so far, all methods had run into issues. This [article](https://medium.com/@jorge.gongora2610/how-to-set-up-an-apache-web-server-on-azure-using-terraform-f7498daa9d66) gave me a few ideas and I set up the provider block using my own code variation.

Still no dice. The issue I am running into this point is that sudo prompts for a password and I cannot find a way to pass credentials (securely or otherwise) to sudo without breaking continuity of the shell. After the first sudo, the credentials are cached, so I merely need a way to pass credentials the first time and bypass the prompt.

I log into the VM directly and start experimenting within the SSH terminal. I try different ways to bypass the sudo prompt. After much experimentation I discover one way; pass the password via echo into sudo -S, which takes input from the standard out. This is probably not the best way to implement this, but so far, it’s been the only method I can determine in the time I have.

I make the changes and thankfully, the changes take and the Apache install completes successfully.

The website comes up as well.

Graphical user interface, text

Description automatically generated

I test the SSH to the HVM VMs as well:

Text

Description automatically generated

Text

Description automatically generated

Both can receive SSH requests without a problem.

At this point, everything is working properly, and code cleanup, formatting, and optimization can take place. With the amount of time I have left, optimization is unlikely, but sanitizing the code and formatting for legibility is a minimum. I also comment where appropriate.

### Notes for improvement:

There are a few areas in which this configuration could be improved; code maintainability and usability could be improved with the use of modules and variables. Time constraints and lack of experience with Terraform and DevOps practices contributed to the use of simpler, but less portable/maintainable code.

Security within the configuration should be improved by using a more secure authentication method. Currently, the service principal uses clear text and the SSH logins also use clear text for some passwords. An SSH certificate does exist for all the VMs and is the preferred method, but was the password was implemented to allow the providers to run without incident.

A better method for implementing the Apache environment on the Apache VM. The remote-exec provider requires SSH access from the VM running Terraform. This is not an ideal implementation as script becomes dependent on network access from the remote machine. Furthermore, passing the bash commands via remote-exec is cumbersome and likely prone to breakage.

Nomenclature is a bit inconsistent, and a better standard should be implemented.

There is certainly room for improvement, but the implementation works and can be readily improved with some refactoring.

## Useful Links

Github Repo: <https://github.com/LucidDescent/Lab>

Microsoft Knowledge Base: <https://docs.microsoft.com/en-us/azure/developer/terraform/>

Terraform Azure Registry: <https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>