# Terraform: Automated deployment and management of virtual machines in Azure

## Document version control

| | | | |
|---|---|---|---|
| Name of document: Terraform: Automated deployment and management of virtual machines in Azure | | | Dover IT Hosting & Storage |
| Build version: 1.0 | | | |
| Document Definition: This document is a how-to guide on virtual machine deployment and management in Azure using Terraform. It apprises readers about deploying and managing single/multiple VMs in Azure using Terraform. | Document ID: | | |
| Author: Lakshay Sethi | Effective Date: | 1st June 2022 | |
| Reviewed by: Abdul Razik | | | |

## Document history

| Version | Date | Author | Section | Changes |
|---|---|---|---|---|
| 0.1 | 24th May 2022 | Lakshay Sethi | Complete | Initial Draft |
| 1.0 | 31st May 2022 | Lakshay Sethi | Complete | Final Draft |

# Table of Contents

# List of figures

## Definitions and abbreviations

| Abbreviation | Explanation |
|---|---|
| **VM** | Virtual Machine |
| **CLI** | Command Line Interface |
| **VS Code** | Visual Studio Code |
| **vnet** | Virtual network |
| **rg** | Resource group |
| **nic** | Network Interface Card |

## Scope

The objective of our project was to find a method for the automated delivery of cloud resources. This report is a step in the same direction. It presents users with a process to deploy and manage virtual machines in Azure using Terraform. This report will guide you through the process to deploy and manage single/multiple virtual machines in Azure using Terraform.

## Not in Scope

Azure as a cloud service provider provides us with several resources. This report does not guide readers to deploy and manage all those resources using Terraform, instead it guides you to manage only VMs using Terraform. It also does not provide a method to deploy a custom image VM in Azure using Terraform.

## Setting up the Development Environment

The following steps are needed to be followed before starting any development of Terraform scripts:

1. **Download and install Microsoft visual studio code**: A comprehensive guide on how to install visual studio code can be found at Microsoft visual studio code. Once you have installed VS Code, you can install the Azure Account extension and Hashicorp Terraform extension. These extensions are optional.
2. **Installing Terraform CLI**: Terraform installation can be as simple as downloading a single zip file and installing it by unzipping it to a folder. The zip file contains a single binary file. The folder should be added to the PATH environment variable such that the Terraform binary can be executed from command line without providing its complete path. A comprehensive guide on how to install Terraform on your system can be found at Install Terraform.
3. **Install Azure CLI**: The Azure CLI is a cross-platform command-line tool to connect to Azure and execute administrative commands on Azure resources. The Azure CLI is available to install in Windows, macOS, and Linux environments. A comprehensive guide on how to install Azure CLI on your system can be found at Install Azure CLI

## Virtual Machine in Azure

*Figure 1) Azure Virtual Machine*

Virtual machines are software emulations of physical computers. They include a virtual processor, memory, storage, and networking resources. VMs host an operating system, and you can install and run software just like a physical computer. When using a remote desktop client, you can use and control the VM as if you were sitting in front of it.

With Azure Virtual Machines, you can create and use VMs in the cloud. Virtual Machines provide infrastructure as a service (IaaS) and can be used in different ways. When you need total control over an operating system and environment, VMs are an ideal choice. Just like a physical computer, you can customize all the software running on the VM. This ability is helpful when you're running custom software or custom hosting configurations.

## Terraform

HashiCorp Terraform is an infrastructure as code tool that lets you define both cloud and on-prem resources in human-readable configuration files that you can version, reuse, and share. A complete extensive guide on Terraform can be found at Terraform summarized. Now that we understand what Terraform is and how it works, let's get into the process of VM deployment using it.

## Single Windows VM deployment using Terraform

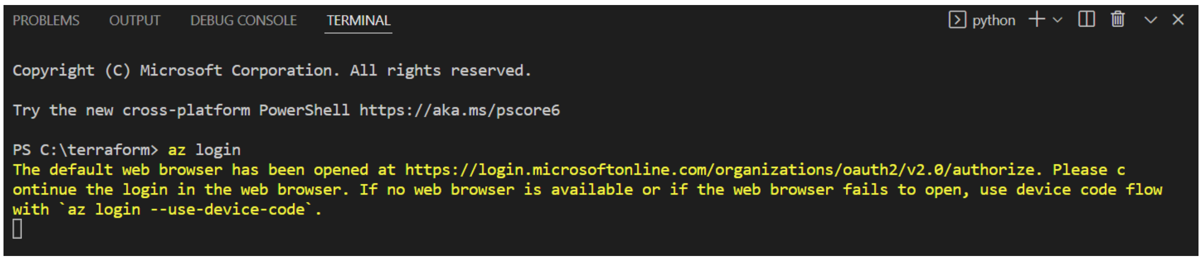
### Authenticate and Authorize Terraform with Azure

Once we have our development environment set up, next we need to authorize and authenticate Terraform with Azure so that it can perform actions on our behalf. There are multiple ways for authentication:

- Authenticating using the Azure CLI
- Authenticating using the service principal and client secret
- Authenticating using the service principal and client certificate
- Authenticating using managed identities

The simplest authentication and authorization method is authentication using the Azure CLI with user credentials. Let's look at how that can be done:

To authenticate to Azure using the `az` CLI command with user credentials, open a terminal window and execute the command `az login`.



```
Copyright (C) Microsoft Corporation. All rights reserved.

Try the new cross-platform PowerShell https://aka.ms/pscore6

PS C:\terraform> az login
The default web browser has been opened at https://login.microsoftonline.com/organizations/oauth2/v2.0/authorize. Please c
ontinue the login in the web browser. If no web browser is available or if the web browser fails to open, use device code flow
with `az login --use-device-code`.
```

*Figure 2) Running az login command*

This will open a new browser window asking the user to log in to Azure with their credentials.

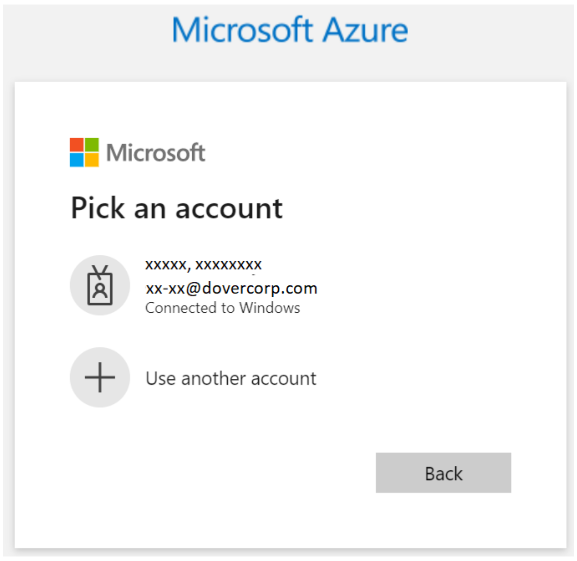
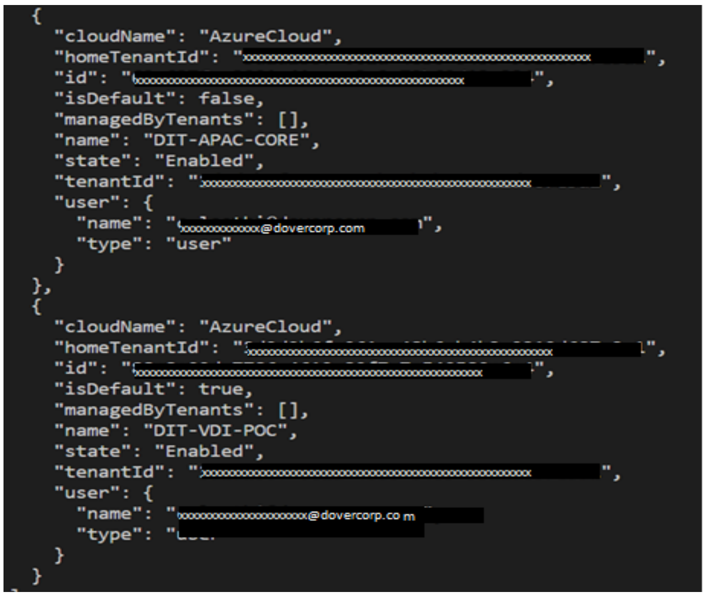


*Figure 3) Login to Azure*

Once the user is logged in, the terminal window will list all the subscriptions that the user has access to.

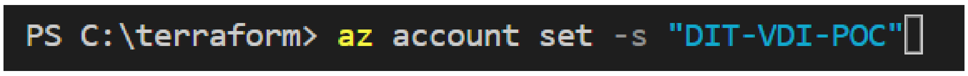


```
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx[redacted]",
    "id": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx[redacted]",
    "isDefault": false,
    "managedByTenants": [],
    "name": "DIT-APAC-CORE",
    "state": "Enabled",
    "tenantId": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx[redacted]",
    "user": {
      "name": "xxxxxxxxxxxx@dovercorp.com[redacted]",
      "type": "user"
    }
  },
  {
    "cloudName": "AzureCloud",
    "homeTenantId": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx[redacted]",
    "id": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx[redacted]",
    "isDefault": true,
    "managedByTenants": [],
    "name": "DIT-VDI-POC",
    "state": "Enabled",
    "tenantId": "xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx[redacted]",
    "user": {
      "name": "xxxxxxxxxxxxxxxxxxxx@dovercorp.co m[redacted]
      "type": "[redacted]
    }
  }
```

*Figure 4) List of subscriptions*

## Selecting operational Subscription

Next, we need to know in which subscription we want to add/update/remove resources and then select that subscription using the command `az account set -s "xxxxxxx-xxxx-xxxxx-xxxxxxxxxxxx"`



```
PS C:\terraform> az account set -s "DIT-VDI-POC"
```

*Figure 5) Command to set working subscription*

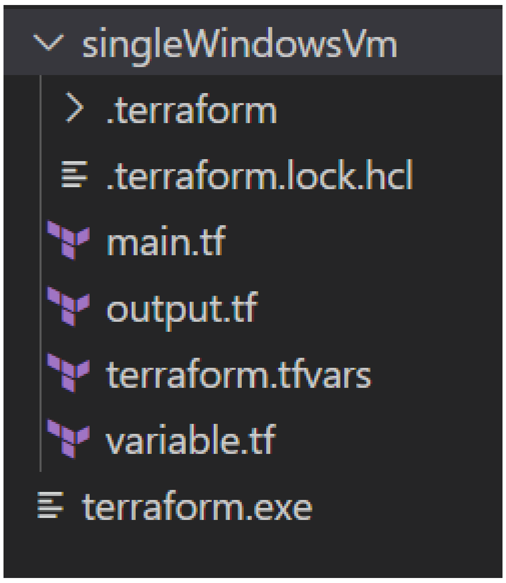
Here, the "s" parameter expects a subscription ID or a subscription name.

Now using the terminal, Terraform commands can be executed, and the remote cloud infrastructure will be managed in the selected subscription.

One can find a tutorial on how to authenticate Terraform using other mentioned ways at Authenticate Terraform.

## Working directory structure

Once we have authorized and authenticated Terraform and selected the subscription, the next step for any cloud resource deployment using Terraform is writing a configuration file that describes that resource. We can write a single configuration file that contains all the variables, resources, data resources, outputs, etc., but that's going to look unwieldy and clumsy. So breaking down a large monolith Terraform file into smaller manageable files is far better and ideal file structure compared to a single file. The script becomes much more comprehendible, easier to read, and helps in faster code editing. So the working directory generally contains the following files:



```
singleWindowsVm
    .terraform
    .terraform.lock.hcl
    main.tf
    output.tf
    terraform.tfvars
    variable.tf
terraform.exe
```

*Figure 6) Working directory structure*

- **Variables.tf**: This file contains the variables declaration.
- **Versions.tf**: This file contains the terraform configuration related to provider and Terraform versioning.
- **Providers.tf**: This file contains the provider configuration.
- **Locals.tf**: This file contains the definitions of local variables.

- **main.tf**: This file contains resource and module configurations. This file can further be broken into multiple files, each containing different sets of resources. We will look into this pattern later in the next chapter.
- **Output.tf**: This file contains the output blocks.
- **terraform.tfvars**: This file is used to provide values to a variable, which otherwise can also be provided at the console.

## Preparing Script file

Now that we know what the folder structure looks like, let's create one to deploy single windows VM:

For a VM we need to know details about the following resources: resource group, virtual network, subnet, and network interface card.

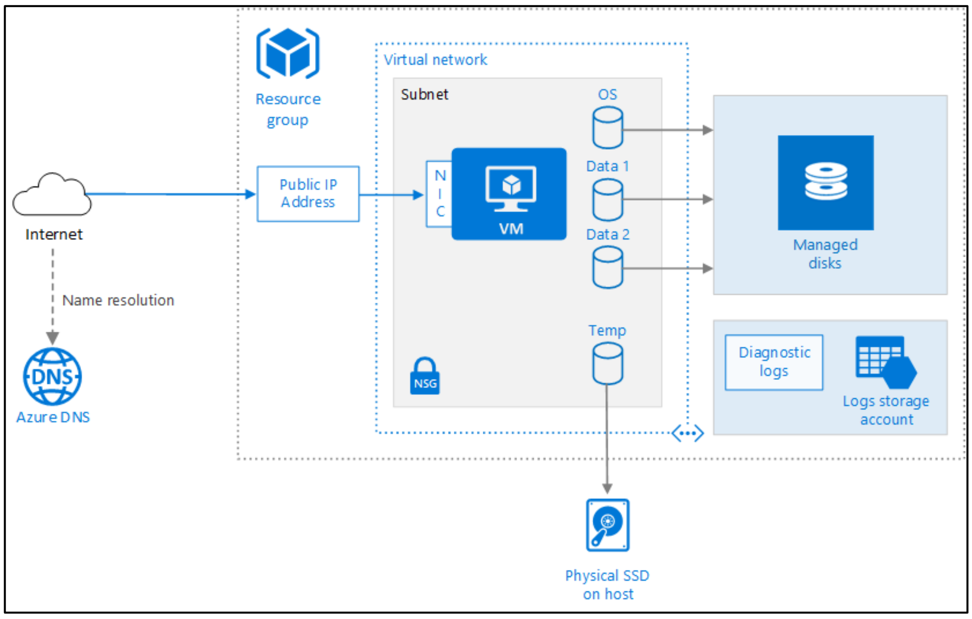
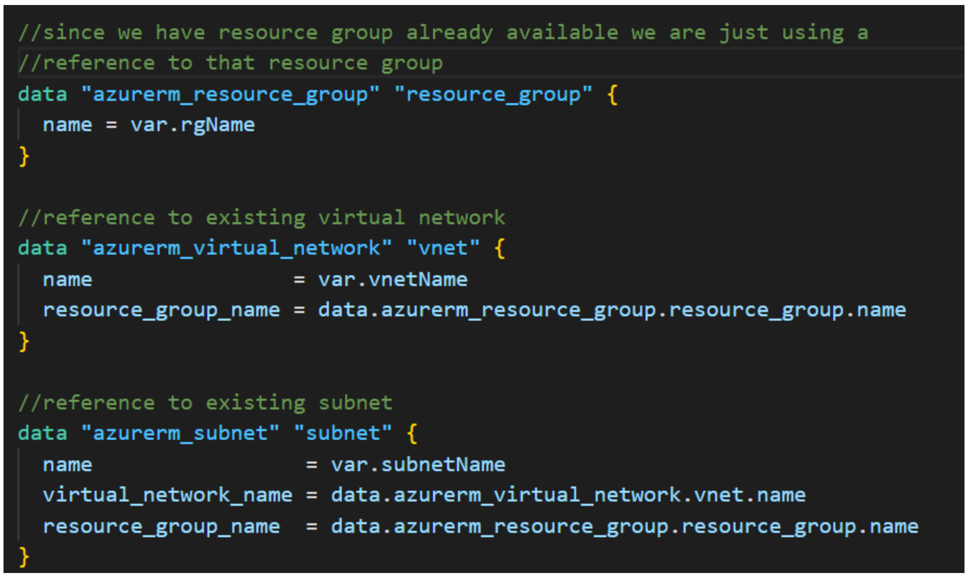


*Figure 7) Resource required to deploy a VM*

For our use case, we had resource group, vnet, and subnet already deployed using the Azure portal. Now to use resources deployed using other means than Terraform, we have data sources in Terraform. Data sources allow Terraform to use the information defined outside of Terraform, defined by another separate Terraform configuration, or by some other means. Now we can point to that rg, vnet, and subnet as:



```
//since we have resource group already available we are just using a
//reference to that resource group
data "azurerm_resource_group" "resource_group" {
  name = var.rgName
}

//reference to existing virtual network
data "azurerm_virtual_network" "vnet" {
  name                = var.vnetName
  resource_group_name = data.azurerm_resource_group.resource_group.name
}

//reference to existing subnet
data "azurerm_subnet" "subnet" {
  name                 = var.subnetName
  virtual_network_name = data.azurerm_virtual_network.vnet.name
  resource_group_name  = data.azurerm_resource_group.resource_group.name
}
```

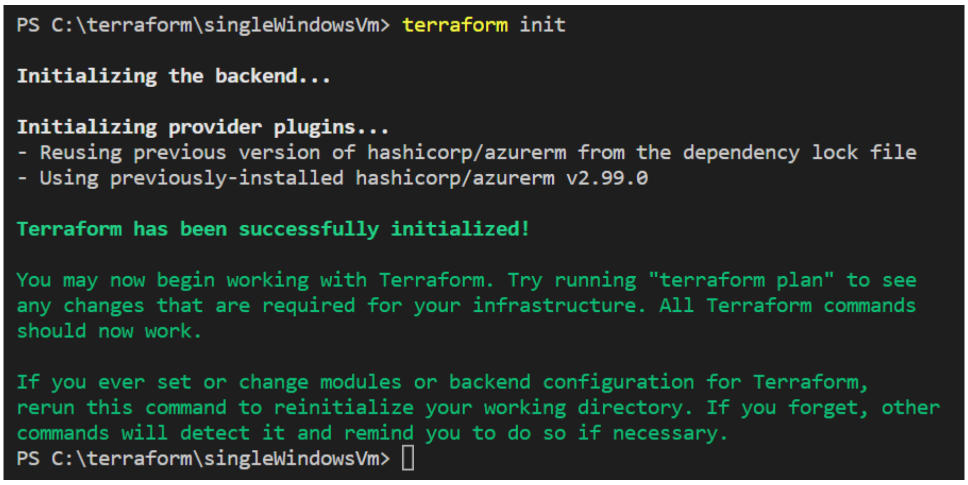
*Figure 8) Data sources in Terraform*

Then we created a new nic for windows VM and an instance of that VM. A complete script to deploy a single Windows VM can be found at singleWindowsVm.

## Deployment Workflow

Once we have our script ready, now to deploy our resource we need to follow the following workflow. This workflow is elaborated in Terraform summarized.

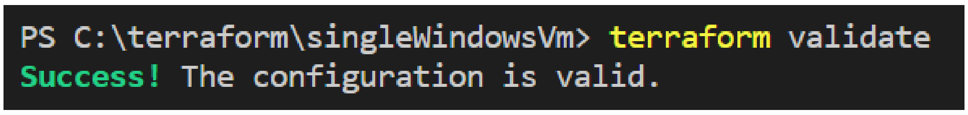
1. Initialize the Terraform working environment in a folder containing the configuration script using `terraform init` command.



```
PS C:\terraform\singleWindowsVm> terraform init

Initializing the backend...

Initializing provider plugins...
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Using previously-installed hashicorp/azurerm v2.99.0

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
PS C:\terraform\singleWindowsVm>
```

*Figure 9) Running init command*

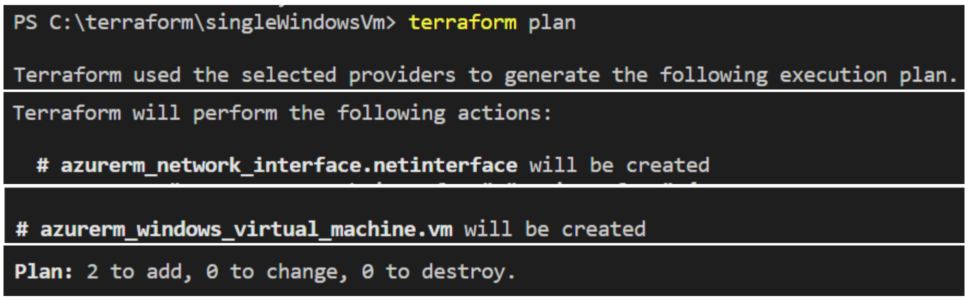
2. Validate the configuration files for syntactical validity and consistency using `terraform validate` command.



```
PS C:\terraform\singleWindowsVm> terraform validate
Success! The configuration is valid.
```

*Figure 10) Running validate command*

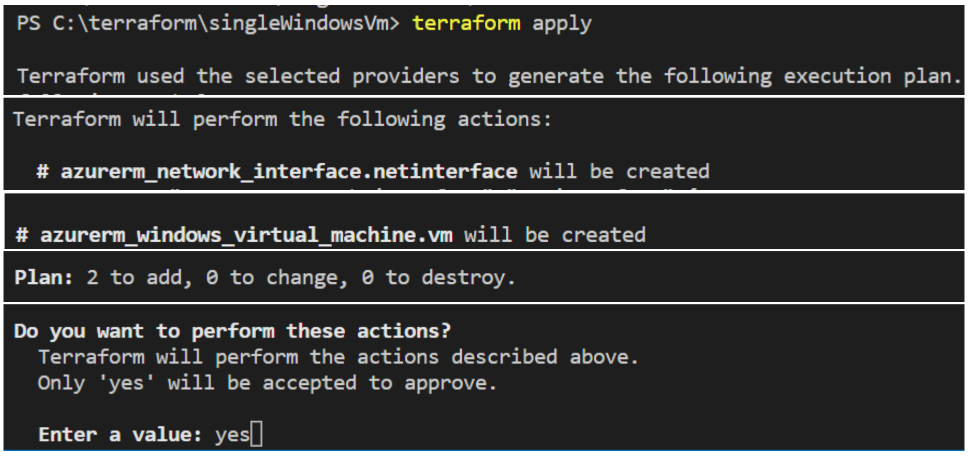
3. Generate the execution plan to check what will change in the target environment using `terraform plan` command



```
PS C:\terraform\singleWindowsVm> terraform plan

Terraform used the selected providers to generate the following execution plan.
Terraform will perform the following actions:

  # azurerm_network_interface.netinterface will be created
# azurerm_windows_virtual_machine.vm will be created
Plan: 2 to add, 0 to change, 0 to destroy.
```

*Figure 11) Running plan command*

4. Apply the configuration to the target environment using `terraform apply` command.



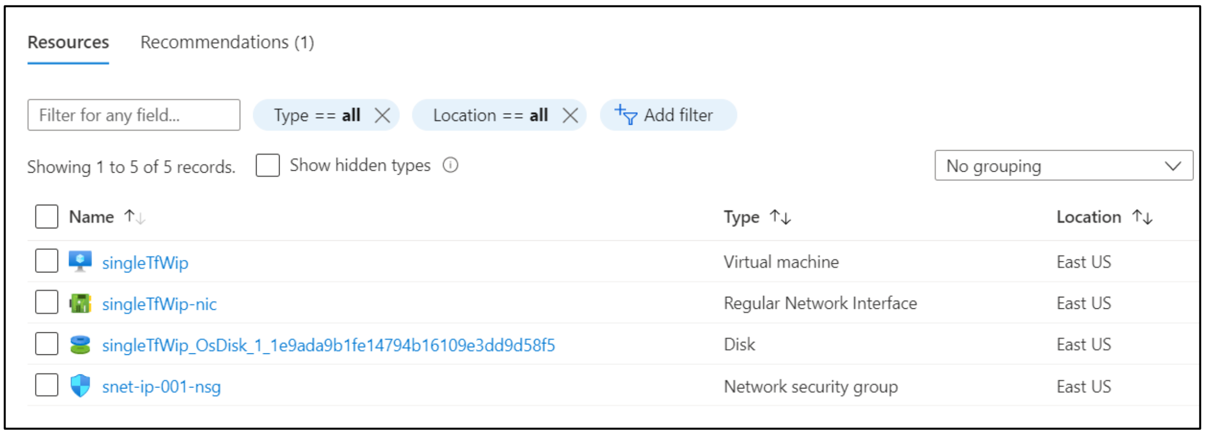
```
PS C:\terraform\singleWindowsVm> terraform apply

Terraform used the selected providers to generate the following execution plan.
Terraform will perform the following actions:

  # azurerm_network_interface.netinterface will be created
# azurerm_windows_virtual_machine.vm will be created
Plan: 2 to add, 0 to change, 0 to destroy.
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
```

*Figure 12) Running apply command*

The template code for any Azure cloud resource can be found at Terraform Registry.

## Output

After applying the Terraform script we can validate the script by checking the Azure portal.



Showing 1 to 5 of 5 records.

| Name | Type | Location |
|---|---|---|
| singleTfWip | Virtual machine | East US |
| singleTfWip-nic | Regular Network Interface | East US |
| singleTfWip_OsDisk_1_1e9ada9b1fe14794b16109e3dd9d58f5 | Disk | East US |
| snet-ip-001-nsg | Network security group | East US |

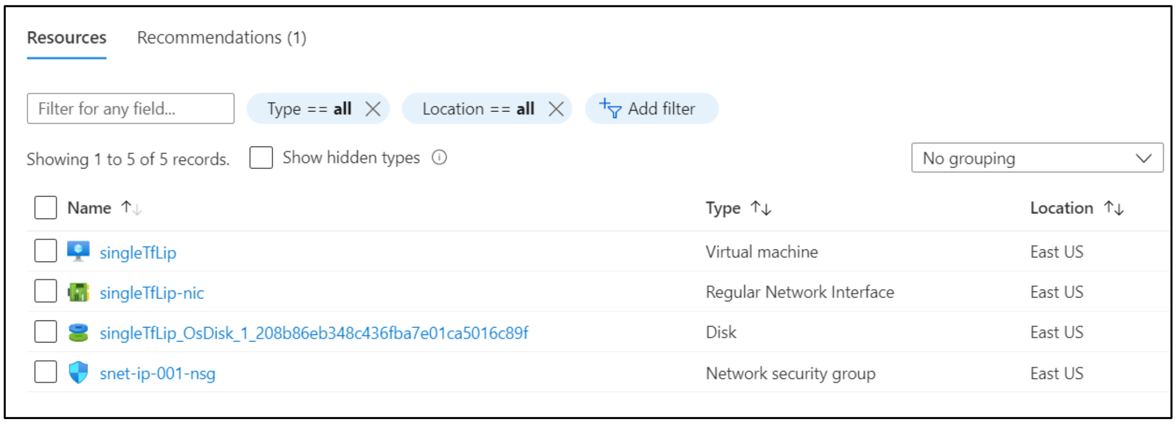
*Figure 13) Azure portal screenshot showing an instance of Windows virtual machine deployed using Terraform*

## Single Linux VM deployment using Terraform

To deploy a single Linux VM using Terraform we will follow a similar workflow as single windows VM deployment using Terraform. The workflow is as follows:

1. Authenticate and authorize Terraform with Azure
2. Select operating subscription
3. Organize working directory structure
4. Prepare script file

   For Linux VM we need almost similar resources as windows VM. We need to know details of the resource group, vnet, subnet, and VM nic. For our use case, we deployed the VM in the same resource group as of windows VM, in the same vnet, but in a different subnet. We provided these details to Terraform using data sources. We created a new nic for the Linux VM and an instance of that VM. A complete script to deploy a single Linux VM for our use case can be found at singleLinuxVm.
5. Apply the script
6. Output



Showing 1 to 5 of 5 records. Show hidden types

| Name | Type | Location |
|---|---|---|
| singleTfLip | Virtual machine | East US |
| singleTfLip-nic | Regular Network Interface | East US |
| singleTfLip_OsDisk_1_208b86eb348c436fba7e01ca5016c89f | Disk | East US |
| snet-ip-001-nsg | Network security group | East US |

*Figure 14) Azure portal screenshot showing an instance of Linux virtual machine deployed using Terraform*

## Multiple Windows VMs deployment using Terraform

To deploy multiple instances of a cloud resource in Terraform using a single script and running that script just once we need to know about Terraform modules, looping elements like count and for_each, and collection and structural datatypes of Terraform. If one knows about these things beforehand, then he/she can skip to workflow to deploy multiple windows VMs.

### Terraform Modules

To create a resource in Azure, we need to either create some required resources or know the details of those resources. For example, for a VM we need a rg, vnet, subnet, and a nic. Now to provision multiple types of environments like development, production, testing, etc, we need to club these multiple compound resources together. So, the module is the logical container of Terraform, which is used to group multiple resources. More details on the module, and how to structure a module can be found at Terraform summarized. A usage example of a module is shown in Figure 16 and Figure 18.

### Looping Elements

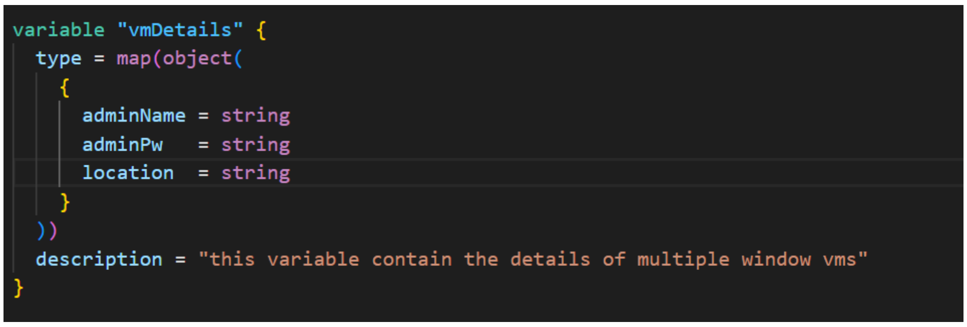
To create multiple instances of a resource using Terraform, one way is to recurrently define that resource. But this solution has some major drawbacks, especially around maintainability and upgradability. Any change would have to be replicated across all such resources and increases the overall chances for error despite considerable testing. The configuration files can also

become quite large and nonreadable. So, to overcome these problems Terraform provides two meta elements count and for_each. More details on these meta elements and how to use them for modules or resources can be found at Terraform summarized. A usage example of looping elements is shown in Figure 16 and Figure 18.
One can read about the collection and structural datatype at Terraform summarized. A usage example of collection and structural datatype is shown in Figure 15.

## Workflow to deploy multiple Windows VMs

1. Authenticate and authorize Terraform with Azure
2. Select operating subscription
3. Organize working directory structure
4. Prepare script file

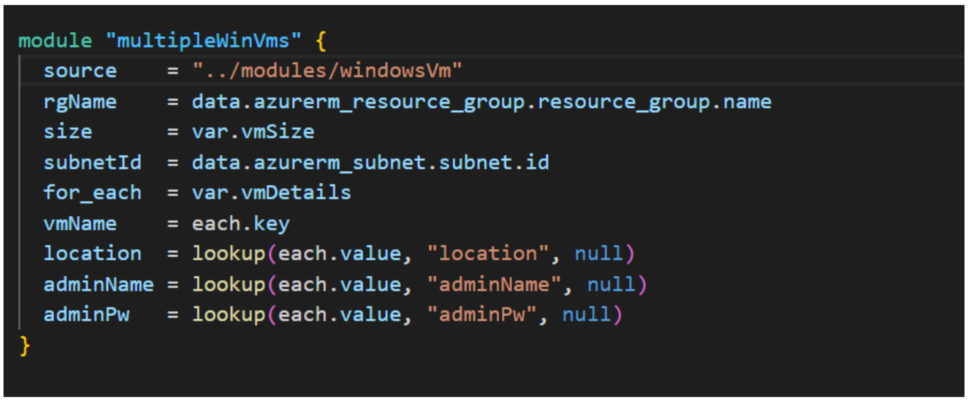
To deploy multiple instances of windows VM, we need to group the required resources in a module. For our use case, we will be deploying all the VMs in the same resource group, in the same vnet and subnet, therefore we don't need to put these resources in a module, instead, we just need to pass details of these resources as input parameters. But we need to create a nic for every VM and an instance of VM, therefore we need to put them in a module. Module for our use case can be found at windowsVm.
In this case, we are providing a different name, different location, different username, and different password for every VM. To do so we have created a map of type object. We have mapped each VM name with an object of type {adminName, adminPw, location}.



```
variable "vmDetails" {
  type = map(object(
    {
      adminName = string
      adminPw   = string
      location  = string
    }
  ))
  description = "this variable contain the details of multiple window vms"
}
```

*Figure 15) Declaring variable of type map of object*

To deploy our use case, we made use of for_each. Configuration details that are common to every single VM are provided to module as input parameter before for_each element, then we provide those detail which varies for every VM. We use each.key to access the VM name and provide that to module, then with each.value we access the mapped object with that VM name and provide the rest of the details.

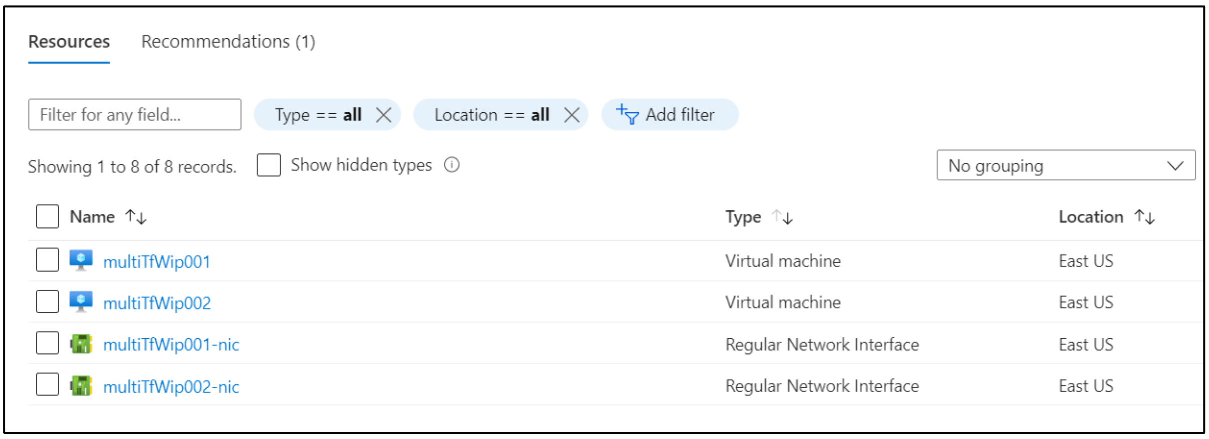


```
module "multipleWinVms" {
  source    = "../modules/windowsVm"
  rgName    = data.azurerm_resource_group.resource_group.name
  size      = var.vmSize
  subnetId  = data.azurerm_subnet.subnet.id
  for_each  = var.vmDetails
  vmName    = each.key
  location  = lookup(each.value, "location", null)
  adminName = lookup(each.value, "adminName", null)
  adminPw   = lookup(each.value, "adminPw", null)
}
```

*Figure 16) Using for_each with windows module to deploy multiple instances of Windows VM*

A complete script to deploy multiple windows VMs for our use case can be found at multipleWindowsVms.

5. Apply the script
6. Output


Resources   Recommendations (1)

Filter for any field...   Type == all   Location == all   Add filter

Showing 1 to 8 of 8 records.   Show hidden types   No grouping

| Name | Type | Location |
|---|---|---|
| multiTfWip001 | Virtual machine | East US |
| multiTfWip002 | Virtual machine | East US |
| multiTfWip001-nic | Regular Network Interface | East US |
| multiTfWip002-nic | Regular Network Interface | East US |

*Figure 17) Multiple instances of Windows VM deployed using Terraform*

## Multiple Linux VMs deployment using Terraform

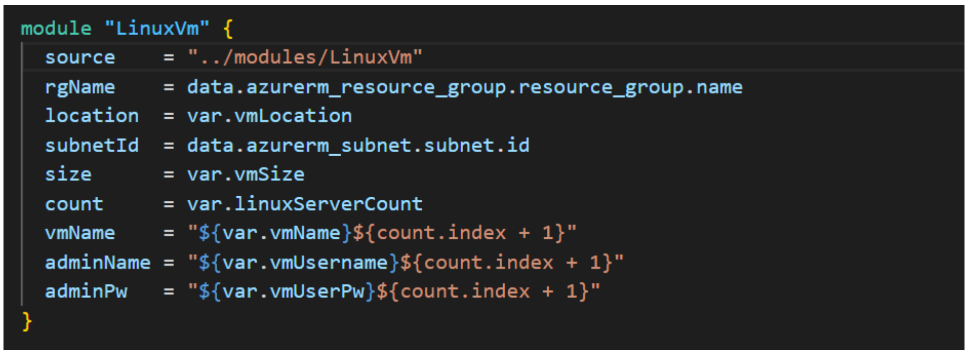
To deploy multiple Linux VM instances using a single Terraform script, we need to follow a similar approach as we followed to deploy multiple windows VMs. This time we will try to implement our logic using count meta element.

### Workflow to deploy multiple Linux VMs

1. Authenticate and authorize Terraform with Azure
2. Select operating subscription
3. Organize working directory structure
4. Prepare script file

In this use case, we will deploy VMs in the same resource group, in the same vnet and subnet, so we will keep these resources in the main file, while for every VM we will create a new nic and its instance, so we will keep them in a module. Module for this use case can be found at linuxVm.

In this use case we will give a common name to all the VMs and just append 1,2,3,4,.....and so on after each VM name, adminName, and adminPw. To do so we will use the count meta element. To append 1,2,3,4…and so on after each variable, we use count.index to take the current value of count and then using string templating we append these numerals after each variable.

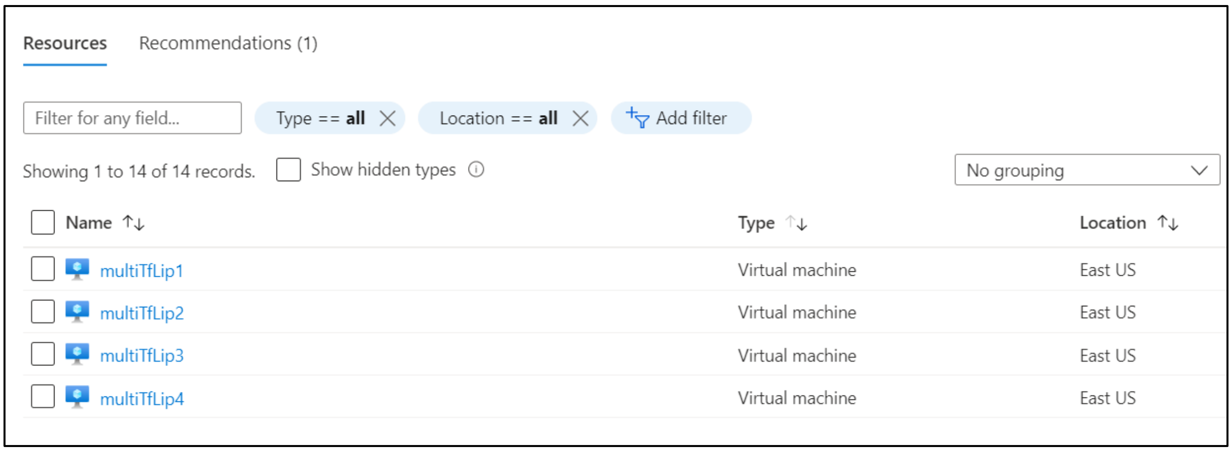


```
module "LinuxVm" {
  source    = "../modules/LinuxVm"
  rgName    = data.azurerm_resource_group.resource_group.name
  location  = var.vmLocation
  subnetId  = data.azurerm_subnet.subnet.id
  size      = var.vmSize
  count     = var.linuxServerCount
  vmName    = "${var.vmName}${count.index + 1}"
  adminName = "${var.vmUsername}${count.index + 1}"
  adminPw   = "${var.vmUserPw}${count.index + 1}"
}
```

*Figure 18) Using count with Linux VM module to deploy multiple instances of Linux VMs*

A complete script to deploy multiple Linux VMs for our use case can be found at multipleLinuxVms.

5. Apply the script
6. Output



| Name | Type | Location |
|---|---|---|
| multiTfLip1 | Virtual machine | East US |
| multiTfLip2 | Virtual machine | East US |
| multiTfLip3 | Virtual machine | East US |
| multiTfLip4 | Virtual machine | East US |

*Figure 19) Multiple instances of Linux VM deployed using Terraform*

## References

[1]. Azure Fundamentals part 1: Describe core Azure concepts (AZ-900) | docs.microsoft.com. Accessed on: April 21, 2022. [Online]. Available: Cloud computing

[2]. Docs overview | hashicorp/azurerm | Terraform registry | registry.terraform.io. Accessed on: May 26, 2022. [Online]. Available: Terraform registry

[3]. Overview of Terraform on Azure – What is Terraform? | Microsoft docs | docs.microsoft.com. Accessed on: April 21, 2022. [Online]. Available: Terraform on Azure

[4]. Infrastructure as code – HashiCorp Terraform | Microsoft Azure | docs.microsoft.com. Accessed on: April 21, 2022. [Online]. Available: Infrastructure as Code

[5]. R. Modi, Deep-Dive Terraform on Azure: Automated Delivery and Deployment of Azure Solutions, 1st ed. Berkeley, CA: Apress, 2021

[6]. Overview – Configuration Language | Terraform by HashiCorp | terraform.io. Accessed on: April 21, 2022. [Online]. Available: HashiCorp Configuration Language

[7]. Install Terraform | Terraform – HashiCorp Learn | learn.hashicorp.com. Accessed on: May 26, 2022. [Online]. Available: Install Terraform

[8]. Setting up Visual Studio Code | code.visualstudio.com. Accessed on: May 26, 2022. [Online]. Available: Microsoft visual studio code

[9]. How to install the Azure CLI | Microsoft Docs | docs.microsoft.com. Accessed on: May 26, 2022. [Online]. Available: Install Azure CLI

[10]. Virtual Machines (VMs) for Linux and Windows | Microsoft Azure | azure.microsoft.com. Accessed on: May 26, 2022. [Online]. Available: Virtual Machines