Managing Microsoft Azure Applications and Infrastructure with Terraform

Course Navigation

Managing Microsoft Azure Applications and Infrastructure with Terraform

Course Introduction

This is an intermediate level course so prior Azure Cloud administration, DevOps knowledge, and concept familiarity is recommended, however, no prior experience with Terraform is neccessary to complete this course. The majority of this course will focus on standard Azure infrastructure resource deployement as well as some basic Azure DevOps operations. Additionally, we will cover a few best-practices to help you use Terraform in a way that will best suit your use requirements.

About the Training Architect

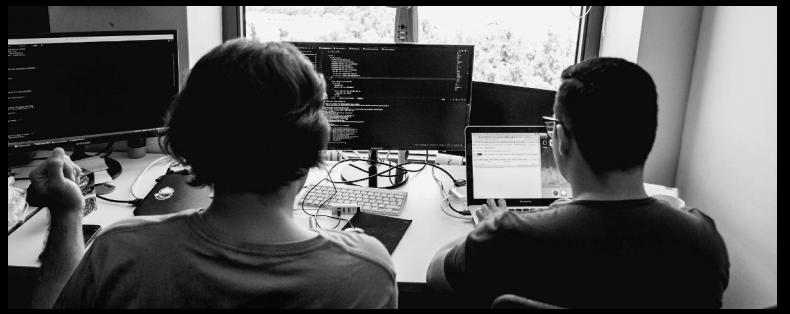
Hi there, my name is Russell Croft and I've been in IT for over 25 years. I began my career as a bench tech, moved quickly into system administration and I've been supporting IT infrastructure at all levels of the enterprise ever since.
I was an IT contractor for many years, working with companies as small as a two person dentist office to multi-national corporate enterprises.
The last fourteen years I've been working in higher education, focussed primarily on infrastrucure and security, with the last few years as a Cloud Architect.
In my career, I've come to value the importance of good training materials and I'm very happy to have the opportunity to help provide the tools and techniques you need to advance in your chosen field.

Thank you for taking this course. Now let's get started!

Russell Croft
Azure Training Architect

Installing Terraform is a pretty straightforward procedure. Just go to https://www.terraform.io/downloads.html and download the version that fits your OS. Unpack the file, add the location to your PATH, and you're ready to go.

It also happens that Azure supports Terraform natively within the Azure CLI, so that's what we're going to focus on for much of this course. Feel free to use whatever text editor you prefer for writing in Terraform's HCL. I generally use VSCode, but plenty of folks out there use Notepad++, VIM, or even one of the text editors native to the CLI itself. Also, make sure you have a resource group and (I recommend) a dedicated storage account in order to use the Azure CLI.



Local State

Pros:

- Only one person making state file changes
- Simpler file location to remember
- Reasonably secure (workstation access only)

Cons:

- State files not easily shared with other admins
- Workstations are more prone to being compromised due to hardware issues or loss of personnel
- Others may have elevated access to a workstation that may not be Azure admins. (Domain admins, desktop support personnel, etc.)
- Workstations are not commonly backed up

Remote State

Pros:

- One source/repository for state files. (Ensures the team is using the same source files for operations)
- Greater security options (encryption, (IAM) role access, restricted network access)
- More options for backups/redundancy
- Less susceptible to hardware or personnel loss
- Allows more users access to Terraform files and allows for version control

Cons:

- Adds complexity to configurations and file access (creation of service principal or managed for access)
- Version control becomes more important/problematic
- Access to remote state files are subject to service outages

Terraform supports a few different methods for Azure authentication:

The simplest and quickest method is to use the **Azure CLI**. Terraform is supported natively by the CLI, and state files live in the storage account used by the account that was used for logging in. By default, this has the benefit of saving your production files in Azure, thereby eliminating many of the concerns of having Terraform files stored locally. However, it also means that each admin logging in to CLI to use Terraform will be doing so with an independent Terraform code base. This isn't really a problem for single-admin shops, one-off deployments for short term environments, or where admins have clearly defined (separate) areas of work. Two or more administrators working in the same space, however, and there can be confusion.

The standard (best practice) for Terraform authentication is to use a **Service Principal**. This allows users and developers to deploy resources via Terraform without their personal accounts needing permission to make changes to the Azure environment. Use of a Service Principle also supports Remote State storage and "backend" features, such as state locking (preventing conflicting changes to occur simultaneously) and remote operations, which allow larger deployments to run independently of the workstation that applied them. When using the Azure CLI or a single workstation, the "backend" is considered to be local.

Managed Service Identities are another way for Terraform to authenticate with Azure. This is a relatively new feature that somewhat simplifies the auth process but still creates a Service Principal to support its functionality. It still has limitations as to what applications are supported, so the extra steps (like key exchange) are generally considered worth it for advanced Terraform users.

For this course, we'll be using the Azure CLI to perform the Terraform operations since the focus is on what Terraform does as its core functionality — deploying infrastructure!

Click here for a link to AzureRM Backend configurations.

An easy way to set up remote state file storage in Azure is to attach it to a blob storage container. An immediate benefit to this is that Azure provides automatic encryption at rest. To set up remote state storage, create a Terraform "backend" statement to designate the remote location of your state file. Just put the resource group, storage account, and storage container name in the **`main.tf`** file in the format below and rerun **`Terraform init`**, answering "yes" to the prompt asking if you wish to copy the local state file to the new location. Make sure to run **`Terraform init`** before updating the **`main.tf`** file with the backend storage configuration. It's likely you have already done this if you used Terraform to create the storage container you intended on using as the "backend" storage for the **`terraform.tfstate`** file. To move the state file back to local storage, simply **#** the Terraform "backend" section you added initially, again answering "yes" to the copy question.

```
# Edit the main.tf for remote state storage

provider "azurerm" {
   version = 1.38
}
terraform {
  backend "azurerm" {
    resource_group_name   = "TFResourceGroup"
    storage_account_name  = "storage4terraform"
    container_name        = "statefile"
    key                   = "terraform.tfstate"
     }
}
```

Click here for a link to AzureRM Backend configurations.

State File Security

Once remote state file storage is in place, you have a number of options to protect your state file data:

Encryption at rest - All Azure blob storage is AES 256 encrypted.

Snapshots of state file data - Routine snapshotting of the state file protects against accidental file deletion.

Apply a Delete Lock to the storage account - Only accounts with "Owner" role access will be able to remove the lock and delete the state file blob. If you ensure that you never perform Terraform activity with an "Owner" account, you'll prevent accidental deletion.

Role Access (IAM) restrictions - If a Service Principle or Managed Service Identity is being used for Terraform activity, you can restrict storage account access to only those accounts. As mentioned above, make sure not to set those accounts with "Owner" access.

Selected Network Access to the Storage Account - If using Terraform from a specific VM or VMs, you can restrict access to only those VNETs and Subnets that contain those VMs. Additionally, you can "whitelist" specific IP addresses both inside and outside your on-premise networks.

Suggestion: Use your preferred GIT Repo or version control software to keep your "working" Terraform files organized and under control. Additionally, you can use Terraform Cloud for version control, found at:
https://www.terraform.io/docs/cloud/index.html

Deploying a Resource Group is a pretty standard procedure. The example below shows the minimum inputs necessary for Resource Group creation.

```
provider "azurerm" {
}
resource "azurerm_resource_group" "rg" {
    name = "TFResourceGroup"
    location = "eastus"
}
```

When you want to add a tag to a resource you're deploying, simply add a **tags** value to the resource as seen in the red box below. You'll use this same format for anything you deploy with Terraform. Remember to **close the brackets** after your tag variables!

If you want to update a resource you've already deployed, you can add the tag variables to the original `.tf` file you used to create it, and run `terraform plan` to ensure your changes will take place and then apply it again.

```
provider "azurerm" {
}
resource "azurerm_resource_group" "rg" {
    name = "TFResourceGroup"
    location = "eastus"

    tags = {
    environment = "Terraform"
    deployedby = "Admin"
  }
}
```

Below is an example of Terraform code that will prompt you for the Region, Resource Group Name, and the Storage Account name you're going to use to create the storage account. The **`variable`** declaration triggers the prompt when you run either **Terraform plan** or **Terraform apply**.

Please note, as long as you have the variable declarations in your **`.tf`** file, Terraform will prompt you for them **every time** you run an **`apply`** job. This can cause you to unintentionally destroy the resources you created the first time you ran **`apply`**. If you use this to create a more permanent resource, you'll want to go back and remove the variable calls and enter the actual names of the resource you created. This format is handy when creating one-off resources such as in dev/test environments as well as when handing over a **`.tf`** file to less technical users.

```
variable "region" {}
variable "ResourceGroup" {}
variable "Storage_Account_Name" {}

resource "azurerm_storage_account" "sa" {
 name                = var.Storage_Account_Name
 resource_group_name     = var.ResourceGroup
 location            = var.region
 account_tier          = "Standard"
 account_replication_type = "GRS"

 tags = {
  environment = "Terraform Storage"
  CreatedBy = "TF Admin"
   }
}
```

Here is a sample of code where you enter in the **resource names** without creating a prompt when **`apply`** is run. This is more appropriate when Terraform is being used to manage infrastructure.

```
resource "azurerm_storage_account" "sa" {
 name                = "Storage Account Name"
 resource_group_name     = "Resource Group"
 location            = "Location/Region"
 account_tier          = "Standard"
 account_replication_type = "GRS"

 tags = {
  environment = "Terraform Storage"
  CreatedBy = "TF Admin"
  }
 }
```

Recovery Service Vaults are the next step after storage account creation. As with storage accounts, you'll only need some basic information to set up deployment.

Below is an example of a Recovery Service Vault referencing a pre-existing resource group in which to deploy the vault.

```
resource "azurerm_recovery_services_vault" "vault" {
  name                = "Terraform-recovery-vault"
  location            = "East US"
  resource_group_name = "TFResourceGroup"
  sku                 = "Standard"
}
```

```
variable "resgrp" {
   description = "Copy and paste the resource group name from the portal."
}
variable "storageaccount"  {
   description = "Enter a unique name for the storage account that will be used for the file share"
}
variable "container" {
   description = "Enter container name"
 }

resource "azurerm_storage_account" "new" {
  name                      = var.storageaccount
  resource_group_name       = var.resgrp
  location                  = "eastus"
  account_tier              = "Standard"
  account_replication_type = "LRS"
}

resource "azurerm_storage_container" "new" {
  name                      = var.container
   storage_account_name     = var.storageaccount
  container_access_type     = "private"
}

resource "azurerm_storage_blob" "new" {
  name                      = "newTFblob"
  resource_group_name      = var.resgrp
  storage_container_name = var.container
  type                      = "Block"
}
```

In this section, we're going to create a 50GB file share as well as some blob storage. If you already know what resource group, storage account, and file share name to use, then you can skip the variable declarations so that the code for the file share would read more like this:

```
resource "azurerm_storage_account" "example" {
  name                      = "awesomestorageaccount"
  resource_group_name       = "resourcegroupforstorage"
  location                  = "eastus"
  account_tier              = "Standard"
  account_replication_type  = "LRS"
}

resource "azurerm_storage_share" "example" {
  name                 = "awesomeshare"
  storage_account_name = azurerm_storage_account.example.name
  quota                = 50
```

Many IT departments will maintain a "template" for VNET/Subnet configurations to allow for rapid deployment of their virtual network infrastructure for infrastructure exapansion or for dev/test environments or even ad hoc networks for demos or UAT, User Acceptance Testing. Just change the network addresses

The only prerequisite for deploying a virtual network (VNET) is a resource group.
Below is an example of a standalone VNET deployment.

```
# Create virtual network
resource "azurerm_virtual_network" "myterraformnetwork"
{
    name                = "myVnet"
    address_space       = ["10.0.0.0/16"]
    location            = "westus"
    resource_group_name = "TFResourceGroup"
}
```

Subnets require an existing VNET with sufficient address space.
Below is an example of a Subnet deployment with a pre-existing VNET.

```
# Create subnet
resource "azurerm_subnet" "tfsubnet" {
    name                 = "mySubnet"
    resource_group_name  = "TFResourceGroup"
    virtual_network_name = "myVnet"
    address_prefix       = "10.0.1.0/24"
}
```

If you wish to create the vnet and subnet together, use the format:
azurerm_virtual_network.TFNet.name (as below)
for the value of "virtual_network_name" and Terraform will ensure that the vnet is created before it continues with the subnet creation. If you were to simply put the first and second examples together, the subnet creation would fail as it would not recognize the vnet dependency.

```
resource "azurerm_virtual_network" "TFNet" {
    name                = "myVnet"
    address_space       = ["10.0.0.0/16"]
    location            = "East US"
    resource_group_name = "TFResourceGroup"
}
resource "azurerm_subnet" "myterraformsubnet" {
    name                 = "mySubnet"
    resource_group_name  = "TFResourceGroup"
    virtual_network_name = azurerm_virtual_network.TFNet.name
    address_prefix       = "10.0.1.0/24"
}
```

Setting up an **NSG** is less about the NSG itself and more about the Inbound/Outbound rule management. Azure doesn't let you renumber network security group inbound/outbound rules easily. You have to delete them and recreate them if you need to renumber them. This has sent many an Admin into fits of rage and has more than once been the justification for integrating more costly software and/or virtual appliances for subnet traffic rules.

If **Terraform** is used to deploy NSG rules, then an Admin can renumber a given set of traffic rules very quickly.

Take the example to the right. Let's suppose we've deployed some inbound rules for web traffic into an NSG, but later discover that we need to insert a rule in between the 1000 and 1001 priority. Simply by re-ordering the priorities on the rules affected by the change and re-running the **Terraform Apply**, the NSG will get updated with minimal impact to the network. It's much easier to edit a **`.tf`** file than to go through what may be hundreds of manual NSG rule changes.

Go to https://www.terraform.io/docs/providers/azurerm/r/network_security_rule.html for the list of optional settings for NSG rules.

```
resource "azurerm_network_security_group" "nsg" {
  name                = "TestNSG"
  location            = "East US"
  resource_group_name = "TFResourceGroup"
}

resource "azurerm_network_security_rule" "example1" {
  name                        = "Web80"
  priority                    = 1001
  direction                   = "Inbound"
  access                      = "Allow"
  protocol                    = "Tcp"
  source_port_range           = "*"
  destination_port_range      = "80"
  source_address_prefix       = "*"
  destination_address_prefix  = "*"
  resource_group_name         = "TFResourceGroup"
  network_security_group_name = azurerm_network_security_group.nsg.name
}

resource "azurerm_network_security_rule" "example2" {
  name                        = "Web8080"
  priority                    = 1000
  direction                   = "Inbound"
  access                      = "Deny"
  protocol                    = "Tcp"
  source_port_range           = "*"
  destination_port_range      = "8080"
  source_address_prefix       = "*"
  destination_address_prefix  = "*"
  resource_group_name         = "TFResourceGroup"
  network_security_group_name = azurerm_network_security_group.nsg.name
}
```

Deploying Azure VMs

```
data "azurerm_subnet" "tfsubnet" {
  name                 = "mySubnet"
  virtual_network_name = "myVnet"
  resource_group_name  = "TFResourceGroup"
}
resource "azurerm_public_ip" "example" {
  name                 = "pubip1"
  location             = "East US"
  resource_group_name  = "TFResourceGroup"
  allocation_method    = "Dynamic"
  sku                  = "Basic"
}
resource "azurerm_network_interface" "example" {
  name                 = "forge-nic"  #var.nic_id
  location             = "East US"
  resource_group_name  = "TFResourceGroup"

   ip_configuration {
    name                          = "ipconfig1"
    subnet_id                     = azurerm_subnet.tfsubnet.id
    private_ip_address_allocation = "Dynamic"
    public_ip_address_id          = azurerm_public_ip.example.id
  }
}
resource "azurerm_storage_account" "sa" {
  name                 = "forgebootdiags123" #var.bdiag
  resource_group_name  = "TFResourceGroup"
  location             = "East US"
  account_tier         = "Standard"
  account_replication_type = "LRS"
  }
```

This section assigns the .id value for the **`azurerm_subnet.tfsubnet`** call during the **`ip_configuration`** section.

Creates a public IP address that can be added to the VM NIC with manual entries for the name, location, and resource group.

This section creates the NIC.

This section sets the IP info for the NIC that is being created. Note that the **`subnet_id`** value calls the name set by the data assignment set at the beginning.

This section creates the boot diagnostic storage account for the VM.

```
resource "azurerm_virtual_machine" "example" {
  name                = "forge"  #var.servername
  location            = "East US"
  resource_group_name   = "TFResourceGroup"
  network_interface_ids = [azurerm_network_interface.example.id]
  vm_size             = "Standard_B1s"
  delete_os_disk_on_termination = true
  delete_data_disks_on_termination = false
  storage_image_reference {
    publisher  = "Canonical"
    offer        = "UbuntuServer"
    sku          = "16.04-LTS"
    version      = "latest"
  }
storage_os_disk {
    name                = "osdisk1"
    disk_size_gb        = "128"
    caching             = "ReadWrite"
    create_option       = "FromImage"
    managed_disk_type = "Standard_LRS"
  }
  os_profile {
    computer_name    = "forge"
    admin_username  = "vmadmin"
    admin_password   = "Password12345!"
  }
  os_profile_linux_config {
    disable_password_authentication = false
  }
boot_diagnostics {
      enabled      = "true"
      storage_uri = azurerm_storage_account.sa.primary_blob_endpoint     }
}
```

Now that the prerequisites set in the previous sections have been created, we can now deploy the actual VM.

```
resource "azurerm_app_service_plan" "svcplan" {
  name                = "newweb-appserviceplan"
  location            = "eastus"
  resource_group_name = "TFResourceGroup"

  sku {
    tier = "Standard"
    size = "S1"
  }
}

resource "azurerm_app_service" "appsvc" {
  name                = "custom-tf-webapp-for-thestudent"
  location            = "eastus"
  resource_group_name = "TFResourceGroup"
  app_service_plan_id = azurerm_app_service_plan.svcplan.id

  site_config {
    dotnet_framework_version = "v4.0"
    scm_type                 = "LocalGit"
  }
}
```

Here we have a deployment for a **dot NET web application**.

Just as via the Azure portal, an app service plan needs to be created for the application, with the **Tier** and **SKU** size indicated as well.

The **`site_config`** section in this example is optional since these settings can be configured after the application service has been deployed. However, the more information you have about how to configure the app, the quicker your developers will be up and running.

Additional information about the optional settings available for application deployments can be found here.

Deploying Database Instances

```
resource "azurerm_mysql_server" "example" {
  name                = "mysql-terraformserver-1"
  location            = "eastus"
  resource_group_name = "TFResourceGroup"

  sku {
    name     = "B_Gen5_2"
    capacity = 2
    tier     = "Basic"
    family   = "Gen5"
  }

  storage_profile {
    storage_mb            = 5120
    backup_retention_days = 7
    geo_redundant_backup  = "Disabled"
  }

  administrator_login          = "mysqladminun"
  administrator_login_password = "easytologin4once!"
  version                      = "5.7"
  ssl_enforcement              = "Enabled"
}

resource "azurerm_mysql_database" "example" {
  name                = "exampledb"
  resource_group_name = "TFResourceGroup"
  server_name         = azurerm_mysql_server.example.name
  charset             = "utf8"
  collation           = "utf8_unicode_ci"
}
```

MariaDB is virtually identical to MySQL. Only the azurerm resource (**azurerm_mariadb_server**) and the version number (as appropriate to MariaDB) are different.

Alert!

Terraform 0.12 does not currently support managed SQL instance creation in Azure. It can only be used to create MySQL databases on existing servers.