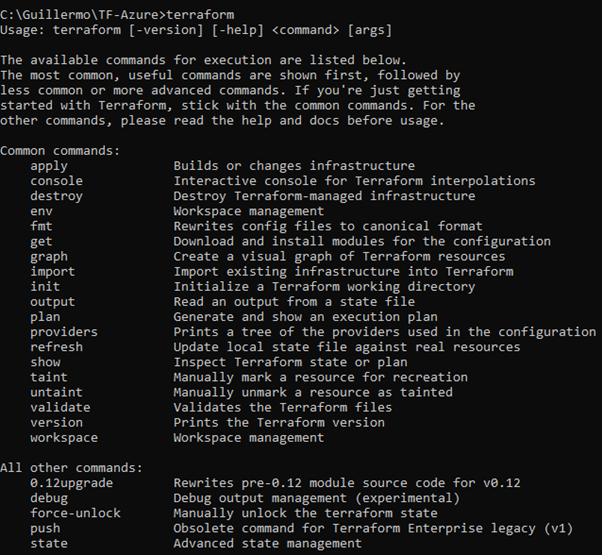
# 1. Installing Terraform

Installing **Terraform** is very simple. We just need to download the file from <https://www.terraform.io/downloads.html> and extract the executable to a folder. Then, we add the folder to the PATH variable in our operating system.

**Terraform** is available for Windows, macOS, Linux, FreeBSD, OpenBSD, and Solaris.

To verify Terraform is working, we type **terraform** in the command prompt.

Image for post

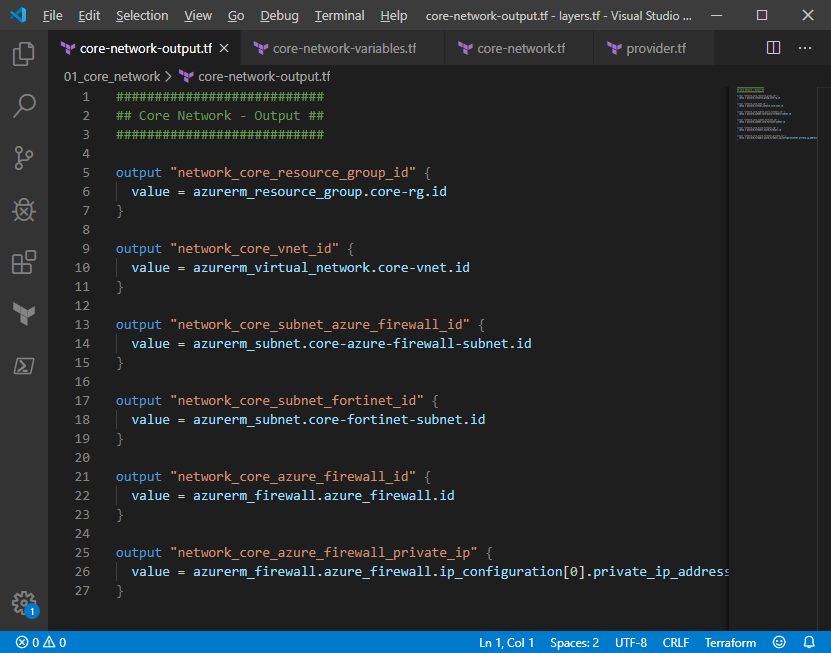


# 2. Preparing to write Terraform code

The most popular editor to write Terraform code is **Visual Studio Code**. It is a free & open-source editor available for Windows, macOS, and multiple versions of Linux. Download it at [https://code.visualstudio.com](https://code.visualstudio.com/)

Then, install the **Terraform extension By Mikael Olenfalk** and the **Azure Terraform extension by Microsoft**.

Image for post



# 3. Terraform Configuration Files

Configuration files can be in two formats:

* **HashiCorp Configuration Language (HCL).** Extension for HCL files is .tf
* **JSON.** The file extension is .tf.json

HCL is the preferred format and the HCL2 version was introduced in Terraform 0.12. **It is recommendable to use HCL2**.

# 4. Resources and Modules

The main purpose of the Terraform language is declaring **resources**. A group of resources can be grouped into a module, which creates a larger unit of configuration. For example, an Azure VNET is a resource.

A resource describes a single infrastructure object, while a module might describe a set of objects and the necessary relationships between them in order to create a higher-level system. For example, a network module can include one Resource Group, a VNET and Subnets resources.

A Terraform configuration consists of a root module, where evaluation begins, along with a tree of child modules created when one module calls another. Any module should include, at a minimum, a **main.tf**, a **variables.tf**, and **outputs.tf** file (if required).

├── README.md  
├── main.tf  
├── variables.tf  
├── outputs.tf

**Note:**This naming convention is recommended, but not enforced. Some people prefer to use the module purpose to name the files, for example: if we have a module for a virtual machine, we can have **vm-main.tf**, **vm-variables.tf** and **vm-output.tf**.

There are no good or bad, it is always a matter of personal convenience or preference.

If we are using nested modules to split up our infrastructure’s required resources, the **main.tf** file holds all our module blocks and any needed resources not contained within our nested modules.

A simple module’s **main.tf** file, without any nested modules, declares all resources within this file.

The **variables.tf** and **outputs.tf** files contain the input variables and output variables declarations. All variables and outputs should include descriptions.

This is an example of a **variable definition**:

variable "core-vnet-cidr" {  
 type = string  
 description = "The CIDR of the Core VNET"  
}

This is an example of an **output definition**:

output "network\_core\_vnet\_id" {  
 value = azurerm\_virtual\_network.core-vnet.id  
 description = "The Core VNET id"  
}

If using nested modules, they should be in a root module’s subdirectory named **modules/**.

├── README.md  
├── main.tf  
├── variables.tf  
├── outputs.tf  
├── ...  
├── modules/  
│ ├── moduleA/  
│ │ ├── README.md  
│ │ ├── variables.tf  
│ │ ├── main.tf  
│ │ ├── outputs.tf  
│ ├── moduleB/  
│ ├── .../

**Note:** when modules are great to simplify the code, however, they reduce the readability of the code. As always, it is a matter of balance and personal preference.

When we need to persist variable values, create a **terraform.tfvars**file and assign variables within this file.

Also, for very large projects and for readability, we can use multiple **.tfvars** files, invoking them from the command line:

terraform apply -var-file="auth.tfvars" -var-file="network.tfvars"

# 5. Terraform Providers

Terraform can create stacks of infrastructure across multiple providers. For example, a single configuration can create infrastructure in both Azure and AWS.

There are dozens of official and community Terraform providers:

* **Terraform official providers,**maintained by HashiCorp <https://www.terraform.io/docs/providers/index.html>,
* **Community providers**, built by the community of Terraform users and vendors <https://www.terraform.io/docs/providers/type/community-index.html>

# 5.1. Azure Provider

The provider block is used to configure the Azure provider (azurerm) and is responsible for creating and managing resources on Azure.

# Configure the Azure Provider  
provider "azurerm" { }

As an option, we can define a specific Azure Provider version using the **version** parameter. The version argument works with all providers. It is used to constrain the provider to a specific version to prevent downloading a new provider that may possibly contain breaking changes.

If the version isn’t specified, Terraform will automatically download the most recent provider.

# Configure the Azure Provider  
provider "azurerm" {  
 version = "=1.30.0"  
}

# 6. Azure Authentication

Terraform supports 4 different methods for authenticating to Azure:

* Authenticating to Azure using the Azure PowerShell and Azure CLI
* Authenticating to Azure using Managed Service Identity
* Authenticating to Azure using a Service Principal and a Client Certificate
* Authenticating to Azure using a Service Principal and a Client Secret

In this post, we will discuss the first (used to start asap) and the last one (used for DevOps tools and for remote backend (explained later in the section 7).

# 6.1. Authenticating using PowerShell

**Azure PowerShell Module** is an easy way to start using Terraform in a minute.

Detailed instructions for the **Azure PowerShell Module** are available at <https://docs.microsoft.com/en-us/powershell/azure/install-az-ps>

To install the PowerShell Azure module for the active user, we type the following command:

Install-Module -Name Az -AllowClobber -Scope CurrentUser

Then, we use the following command to authenticate:

Login-AzAccount

And the following command to get the list of Azure subscriptions:

Get-AzSubscription

We can select the subscription using the following command:

Select-AzSubscription -SubscriptionId <Azure-SubscriptionId>

# 6.2. Authenticating to Azure using the Azure CLI

**Azure CLI** is a simple way to authenticate and start working with Azure as soon as possible.

If we want to run Terraform from our machine or server and use **Azure CLI** for authentication, we need to install **Azure CLI** from <https://docs.microsoft.com/en-us/cli/azure/install-azure-cli>

**Note:** to install Azure CLI in Windows, check the next point.

After we authenticated with Azure CLI, Terraform will use the UPN (User Principal Name) credentials, such a [user@domain.com](http://user@domain.com/).

To authenticate using **Azure CLI**, we need to type:

az login

The process will launch the browser and after the authentication is complete we are ready to go.

Image for post



We will use the following command to get the list of Azure subscriptions:

az account list --output table

We can select the subscription using the following command (both subscription id and subscription name are accepted):

az account set --subscription <Azure-SubscriptionId>

And the following command to show the current Azure subscription:

az account show

# 6.3. Installing or upgrading AZ CLI for Windows

Currently, CLI doesn’t provide an option to upgrade the version of CLI.

Run the following command to install or upgrade AZ CLI to the latest version:

Invoke-WebRequest -Uri <https://aka.ms/installazurecliwindows> -OutFile .\AzureCLI.msi; Start-Process msiexec.exe -Wait -ArgumentList '/I AzureCLI.msi /quiet'

# 6.4. Authenticating to Azure using a Service Principal and a Client Secret

Using Service Principal, also known as SPN, is a best practice for DevOps or CI/CD environments.

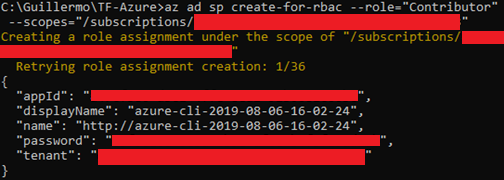
First, we need to authenticate to Azure using **az login**, then select subscription using **az account set** (showed in the previous point).

Then create the service principal account using the following command:

az ad sp create-for-rbac --role="Contributor"   
--scopes="/subscriptions/SUBSCRIPTION\_ID" --name "Azure-DevOps"

This is the result:

Image for post



**Note:** as an option, we can add the -name parameter to add a descriptive name.

These values will be mapped to the Terraform variables -see below-:

* **appId** is the client\_id defined above.
* **password** is the client\_secret defined above.
* **tenant** is the tenant\_id defined above.

Finally, it’s possible to test these values work as expected by first logging in:

az login --service-principal -u CLIENT\_ID -p CLIENT\_SECRET --tenant TENANT\_ID

# 6.5. Configuring the Service Principal in Terraform for Command Line

After we obtained the credentials for the Service Principal, now we can use credentials with variables, to authenticate to Terraform.

First, we define variables in the **variables.tf** file:

# Azure Subscription Id  
variable "azure-subscription-id" {  
 type = string  
 description = "Azure Subscription Id"  
}# Azure Client Id/appId  
variable "azure-client-id" {  
 type = string  
 description = "Azure Client Id/appId"  
}# Azure Client Id/appId  
variable "azure-client-secret" {  
 type = string  
 description = "Azure Client Id/appId"  
}# Azure Tenant Id  
variable "azure-tenant-id" {  
 type = string  
 description = "Azure Tenant Id"  
}

Second, we set variable values in the **terraform.tfvars** file.

**Note: do not store this file in public repositories.**

# Azure Subscription Id  
azure-subscription-id = "9c242362-6776-47d9-9db9-2aab2449703"# Azure Client Id/appId  
azure-client-id = "a47dd585-d5c1-4ffe-8ea2-91f0681c55bc"# Azure Client Secret/password  
azure-client-secret = "88f9776a-0e3h-4b9e-afcc-b4h3f17ad20h"# Azure Tenant Id  
azure-tenant-id = "c7e7beb6-8e7c-41e7-bc41-37ggg3e33b665"

Finally, update the provider section in the **main.tf** to use the Azure variables:

# Configure the Azure Provider  
provider "azurerm" {   
 subscription\_id = var.azure-subscription-id  
 client\_id = var.azure-client-id  
 client\_secret = var.azure-client-secret   
 tenant\_id = var.azure-tenant-id  
}

# 7. What it the Terraform state file?

**Terraform** records information about what infrastructure is created in a Terraform state file (**terraform.tfstate**).

The Terraform state file is used when we run **terraform apply** or **terraform plan** commands and it is used to keep information about the infrastructure created.

Benefits of state files:

* Keep track of infrastructure changes
* Update only required components
* Destroy / rollback of components

Cons of state files:

* The local state does not work well in a team or collaborative environment
* Terraform state can include sensitive information
* Storing state locally increases the chance of inadvertent deletion
* Requires shared storage

**Important: Terraform store credentials in clear text in state files. NEVER store state files in public repositories.**

# 7.1. Where to store the Terraform state file?

For simple test scripts or for development, a local state file will work. However, if **two or more people** need to access or execute the Terraform file, we need to store the file in a remote backend and lock the file to avoid mistakes or damage to existing infrastructure.

Use **remote backends**, such as **Azure Storage, Google Cloud Storage**, **Amazon S3**and **HashiCorp Terraform Cloud & Terraform Enterprise**, to keep our files safe and share between multiple users.

# 7.2. Configure the remote backend to use Azure Storage in Bash or Azure Cloud Shell

Execute the following **Azure CLI script**to create the storage account in Azure Storage in Bash or Azure Cloud Shell:

RESOURCE\_GROUP\_NAME=kopicloud-tstate-rg  
STORAGE\_ACCOUNT\_NAME=kopicloudtfstate$RANDOM  
CONTAINER\_NAME=tfstate# Create resource group  
az group create --name $RESOURCE\_GROUP\_NAME --location "West Europe"# Create storage account  
az storage account create --resource-group $RESOURCE\_GROUP\_NAME --name $STORAGE\_ACCOUNT\_NAME --sku Standard\_LRS --encryption-services blob# Get storage account key  
ACCOUNT\_KEY=$(az storage account keys list --resource-group $RESOURCE\_GROUP\_NAME --account-name $STORAGE\_ACCOUNT\_NAME --query [0].value -o tsv)# Create blob container  
az storage container create --name $CONTAINER\_NAME --account-name $STORAGE\_ACCOUNT\_NAME --account-key $ACCOUNT\_KEYecho "storage\_account\_name: $STORAGE\_ACCOUNT\_NAME"  
echo "container\_name: $CONTAINER\_NAME"  
echo "access\_key: $ACCOUNT\_KEY"

# 7.3. Configure the remote backend to use Azure Storage with PowerShell

Execute the following **Azure PowerShell** script to create the storage account in **Azure Storage**:

# Variables  
$azureSubscriptionId = "9c242362-6776-47d9-9db9-2aab2449703"  
$resourceGroup = "kopicloud-tstate-rg"  
$location = "westeurope"  
$random = -join ((0..9) | Get-Random -Count 5 | % {$\_})  
$accountName = "kopicloudtfstate" + $random  
$containerName = "tfstate"# Set Default Subscription  
Select-AzSubscription -SubscriptionId $azureSubscriptionId# Create Resource Group  
New-AzResourceGroup -Name $resourceGroup -Location $location -Force# Create Storage Account  
$storageAccount = New-AzStorageAccount -ResourceGroupName $resourceGroup `  
 -Name $accountName `  
 -Location $location `  
 -SkuName Standard\_RAGRS `  
 -Kind StorageV2   
   
# Get Storage Account Key  
$storageKey = (Get-AzStorageAccountKey -ResourceGroupName $resourceGroup `  
 -Name $accountName).Value[0]# Create Storage Container  
$ctx = $storageAccount.Context  
$container = New-AzStorageContainer -Name $containerName `  
 -Context $ctx -Permission blob# Results  
Write-Host   
Write-Host ("Storage Account Name: " + $accountName)  
Write-Host ("Container Name: " + $container.Name)  
Write-Host ("Access Key: " + $storageKey)

This is the result:

Image for post

Image for post

# 7.4. Configure the remote backend to use Azure Storage with Terraform

We can also use Terraform to create the storage account in **Azure Storage.**

We create a file called **az-remote-backend-variables.tf** and add this code:

# company  
variable "company" {  
 type = string  
 description = "This variable defines the name of the company"  
}# environment  
variable "environment" {  
 type = string  
 description = "This variable defines the environment to be built"  
}# azure region  
variable "location" {  
 type = string  
 description = "Azure region where the resource group will be created"  
 default = "north europe"  
}

Then we create the **az-remote-backend-main.tf** file that will configure the storage account:

# Generate a random storage name  
resource "random\_string" "tf-name" {  
 length = 8  
 upper = false  
 number = true  
 lower = true  
 special = false  
}# Create a Resource Group for the Terraform State File  
resource "azurerm\_resource\_group" "state-rg" {  
 name = "${lower(var.company)}-tfstate-rg"  
 location = var.location lifecycle {  
 prevent\_destroy = true  
 } tags = {  
 environment = var.environment  
 }  
}# Create a Storage Account for the Terraform State File  
resource "azurerm\_storage\_account" "state-sta" {  
 depends\_on = [azurerm\_resource\_group.state-rg] name = "${lower(var.company)}tf${random\_string.tf-name.result}"  
 resource\_group\_name = azurerm\_resource\_group.state-rg.name  
 location = azurerm\_resource\_group.state-rg.location  
 account\_kind = "StorageV2"  
 account\_tier = "Standard"  
 access\_tier = "Hot"  
 account\_replication\_type = "ZRS"  
 enable\_https\_traffic\_only = true  
   
 lifecycle {  
 prevent\_destroy = true  
 } tags = {  
 environment = var.environment  
 }  
}# Create a Storage Container for the Core State File  
resource "azurerm\_storage\_container" "core-container" {  
 depends\_on = [azurerm\_storage\_account.state-sta] name = "core-tfstate"  
 storage\_account\_name = azurerm\_storage\_account.state-sta.name  
}

Finally, we create the file **az-remote-backend-output.tf** file that will show the output:

output "terraform\_state\_resource\_group\_name" {  
 value = azurerm\_resource\_group.state-rg.name  
}output "terraform\_state\_storage\_account" {  
 value = azurerm\_storage\_account.state-sta.name  
}output "terraform\_state\_storage\_container\_core" {  
 value = azurerm\_storage\_container.core-container.name  
}

# 7.5. Authenticating to Azure using Storage Access Key for State Files in Remote Backend

First, we need to update the Terraform and Azure provider:

# Define Terraform provider  
terraform {  
 required\_version = ">= 0.12"  
 backend "azurerm" {  
 resource\_group\_name = "kopicloud-tfstate-rg"  
 storage\_account\_name = "kopicloudtfstate"  
 container\_name = "core-tfstate"  
 key = "core.kopicloud.tfstate"  
 access\_key = "NZa2wDaIFMo7Hlue/GjVbhaT+x9codEFyIHGhzc  
 /S/yevCvHIDE+BV5rJFgrmKpCz7+/mAF932Zii8TlgE0Ezw=="  
 }  
}# Configure the Azure provider  
provider "azurerm" {   
 environment = "public"  
}

# 7.6. Authenticating to Azure using a Service Principal to use State Files in Remote Backend

If we will need to use shared state files in a remote backend, we can configure Terraform using the following procedure:

We will create a configuration file with the credentials information. For this example, I called the file **azurecreds.conf**. This is the content of the file:

ARM\_SUBSCRIPTION\_ID="9c242362-6776-47d9-9db9-2aab2449703"  
ARM\_CLIENT\_ID = "a47dd585-d5c1-4ffe-8ea2-91f0681c55bc"  
ARM\_CLIENT\_SECRET="88f9776a-0e3h-4b9e-afcc-b4h3f17ad20h"  
ARM\_TENANT\_ID="c7e7beb6-8e7c-41e7-bc41-37ggg3e33b665"

then we update the Terraform and Azure provider:

# Define Terraform provider  
terraform {  
 required\_version = ">= 0.12"  
 backend "azurerm" {  
 resource\_group\_name = "kopicloud-tfstate-rg"  
 storage\_account\_name = "kopicloudtfstate"  
 container\_name = "core-tfstate"  
 key = "core.kopicloud.tfstate"  
 }  
}# Configure the Azure provider  
provider "azurerm" {   
 environment = "public"  
}

Finally, we init the Terraform configuration using this command:

terraform init -backend-config=**azurecreds.conf**

Then, we launch the stack as usual:

terraform apply -auto-approve

# 8. Build infrastructure

Now we are ready to start creating some infrastructure using Terraform.

**Important:** check Azure naming conventions before you start building infrastructure. If you don’t use the right name convention, infrastructure creation will fail.

Refer to → <https://docs.microsoft.com/en-us/azure/architecture/best-practices/naming-conventions>

**Note: Terraform** is installed by default in the **Azure Cloud Shell**. Cloud shell can be run standalone or as an integrated command-line terminal from the Azure portal.

# 8.1. Initialize a Terraform working directory

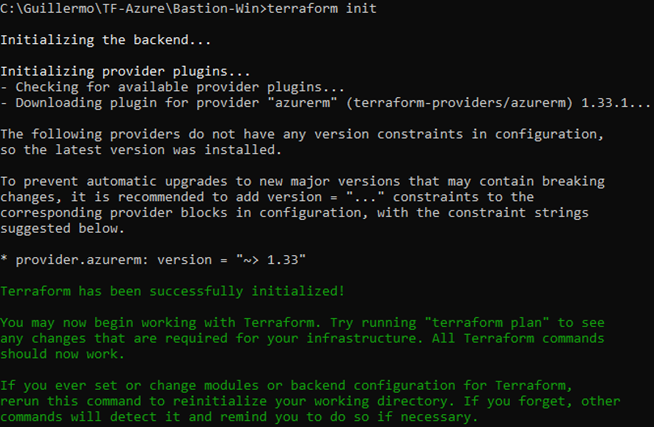
The **terraform init** command is used to initialize a working directory containing Terraform configuration files.

This is the first command that we should be run after writing a new Terraform configuration or cloning an existing one from version control.

This command downloads all required components, including modules and plugins automatically. Also, it initializes the state files.

terraform init

Image for post



# 8.2. Validate your Terraform files

The **terraform validate** command is used to validate the syntax of the terraform files. Terraform performs a syntax check on all the terraform files in the directory and will display an error if any of the files don’t validate.

terraform validate

Image for post

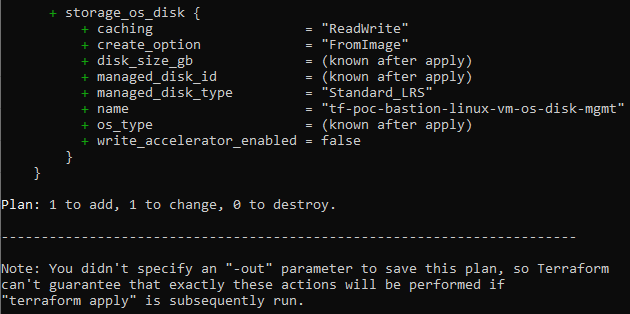
Image for post

# 8.3. Create an execution plan

The **terraform plan** command is used to create an execution plan. Terraform determines what actions are necessary to achieve the desired state specified in the configuration files.

terraform plan

Image for post



# 8.4. Build or change of infrastructure

The **terraform apply** command is used to apply the changes required to reach the desired state of the configuration, or the pre-determined set of actions generated by a terraform plan execution plan.

terraform apply

Image for post

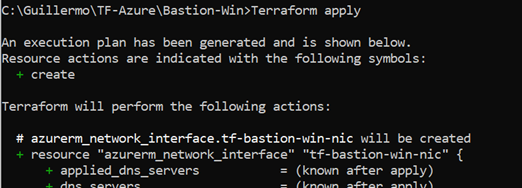
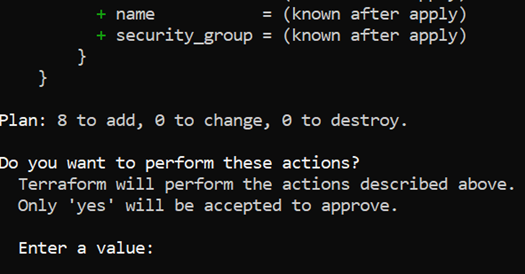
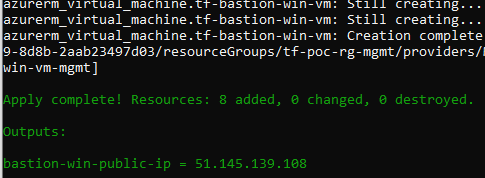


Image for post



Type **yes** to create or update the infrastructure…

Image for post



or use the parameter **-auto-approve** to skip the confirmation.

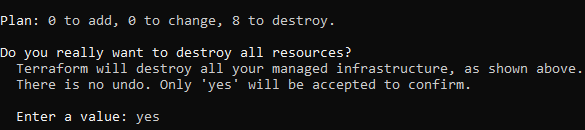
terraform apply -auto-approve

# 8.5. Destroy the infrastructure

The **terraform destroy** command is used to destroy the Terraform-managed infrastructure.

terraform destroy

Image for post



Enter **yes** to delete all resources…