

**Aim: Infrastructure as Code (IaC) for ML:**

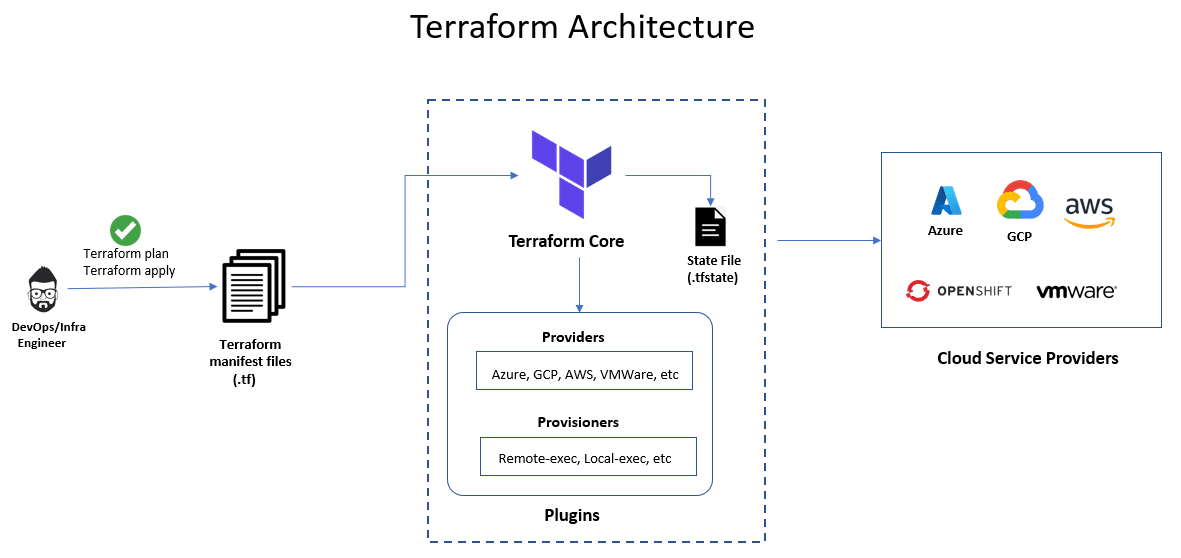
**Use tools like Terraform or AWS CloudFormation to manage ML infrastructure.**

**Experiment with provisioning and automating the setup of ML environments.**

**Theory:**

Terraform, a widely used Infrastructure as Code (IaC) tool. We will provide an overview of Terraform, its objectives, fundamental architecture, and the workflow that encompasses the different stages of provisioning infrastructure with Terraform. We will also highlight the advantages of using Terraform over other cloud-native infrastructure tools.

Terraform is an open-source tool developed by Hashicorp that enables you to provision your infrastructure in a simple, efficient, and declarative manner through repeatable code, whether it be on cloud or on-premises environments.



**Terraform Core**

Terraform’s core (also known as Terraform CLI) is built on a statically-compiled binary that’s developed using the Go programming language.

### Providers

Terraform providers are modules that enable Terraform to communicate with a diverse range of services and resources, including but not limited to cloud providers, databases, and DNS services.

### State file

The [Terraform state file](https://spacelift.io/blog/terraform-state) is an essential aspect of Terraform’s functionality. It is a JSON file that stores information about the resources that Terraform manages, as well as their current state and dependencies.

## Terraform Structure

Declaring resources is very easy in Terraform. [Terraform files](https://spacelift.io/blog/terraform-files) always end with the extension .tf.

The basic Terraform structure contains the following elements.

### Terraform Block

A Terraform block specifies the required providers that terraform needs in order to execute the script. This block also contains the source block that specifies from where terraform should download the provider and also the required version.

Below is an example:

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "=3.0.0"

}

}

}

### Provider Block

A provider block specifies the cloud provider and the API credentials required to connect to the provider’s services. It includes the provider name, version, access key, and secret key.

For example, if you are using Azure as your service provider, it would look as follows:

provider "azurerm" {

features {}

subscription\_id = "00000000-0000-0000-0000-000000000000"

tenant\_id = "11111111-1111-1111-1111-111111111111"

}

### Resource Block

A resource block represents a particular resource in the cloud provider’s services. It includes the resource type, name, and configuration details. This is the main block that specifies the type of resource we are trying to deploy.

Below is an example for creating a resource group in Azure.

resource "azurerm\_resource\_group" "example" {

name = "example"

location = "West Europe"

}

### Data Block

A data block is used to fetch data from the provider’s services, which can be used in resource blocks. It includes the data type and configuration details.

This is used in scenarios where the resource is already deployed, and you would like to fetch the details of that resource.

The code snippet below helps you to fetch details of an existing resource group that is already deployed.

data "azurerm\_resource\_group" "example" {

name = "existing"

}

### Variable Block

A variable block is used to define input variables that are used in the Terraform configuration. It includes the variable name, type, and default value.

Following is an example of a variable block in Terraform.

variable "resource\_group\_name" {

default = "myTFResourceGroup"

}

### Output Block

An output block is used to define [output values](https://spacelift.io/blog/terraform-output) that are generated by the Terraform configuration. It includes the output name and value.

output "resource\_group\_id" {

value = azurerm\_resource\_group.rg.id

}

### Provisioners

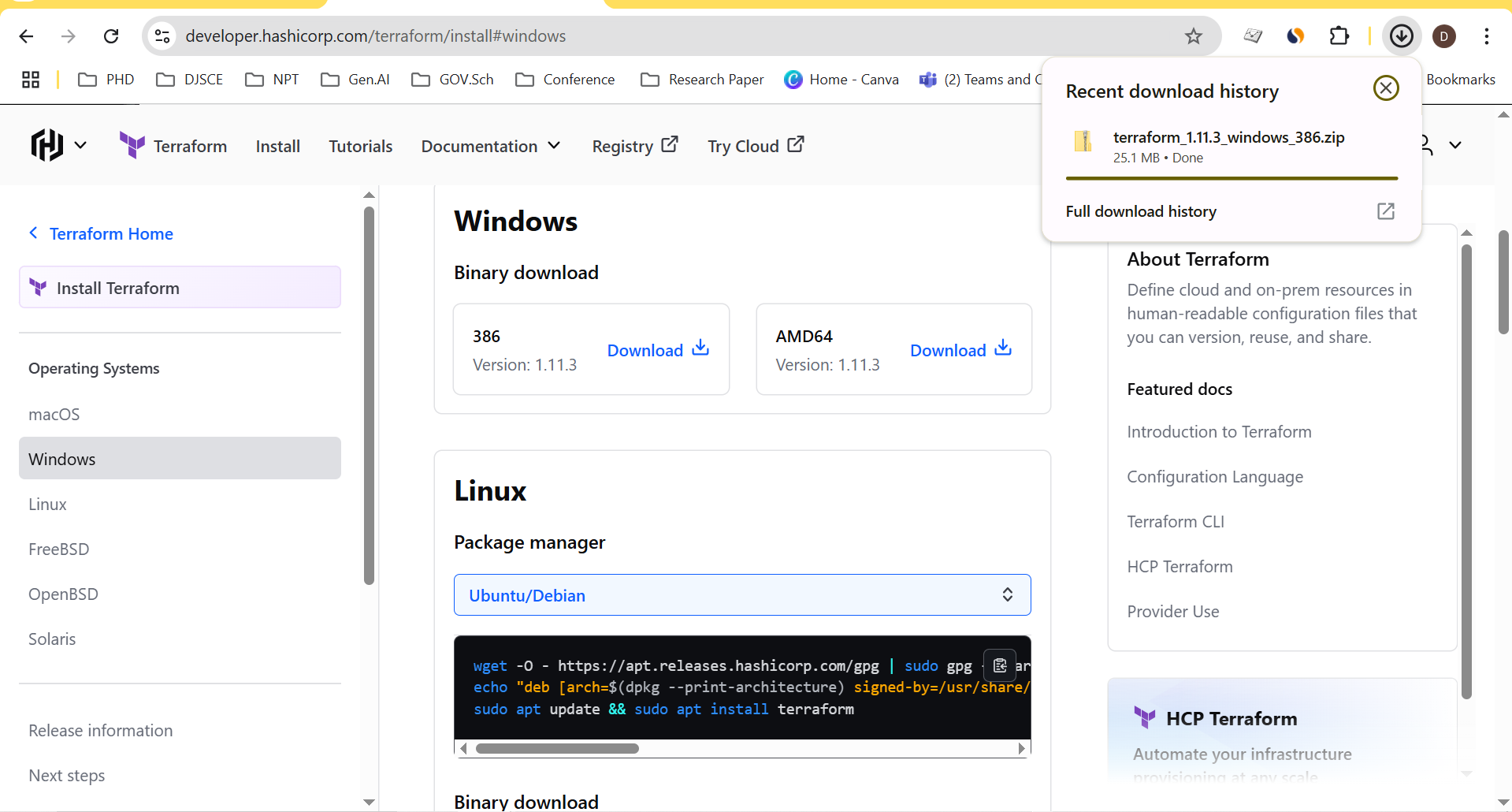
[Terraform provisioners](https://spacelift.io/blog/terraform-provisioners) are a feature that allows Terraform to execute scripts or commands on newly created resources or instances

**Practical (Two terraform scripts available)**

**First Scripts: Terraform configuration to deploy an NGINX container using Docker on Windows (with Docker Desktop)**

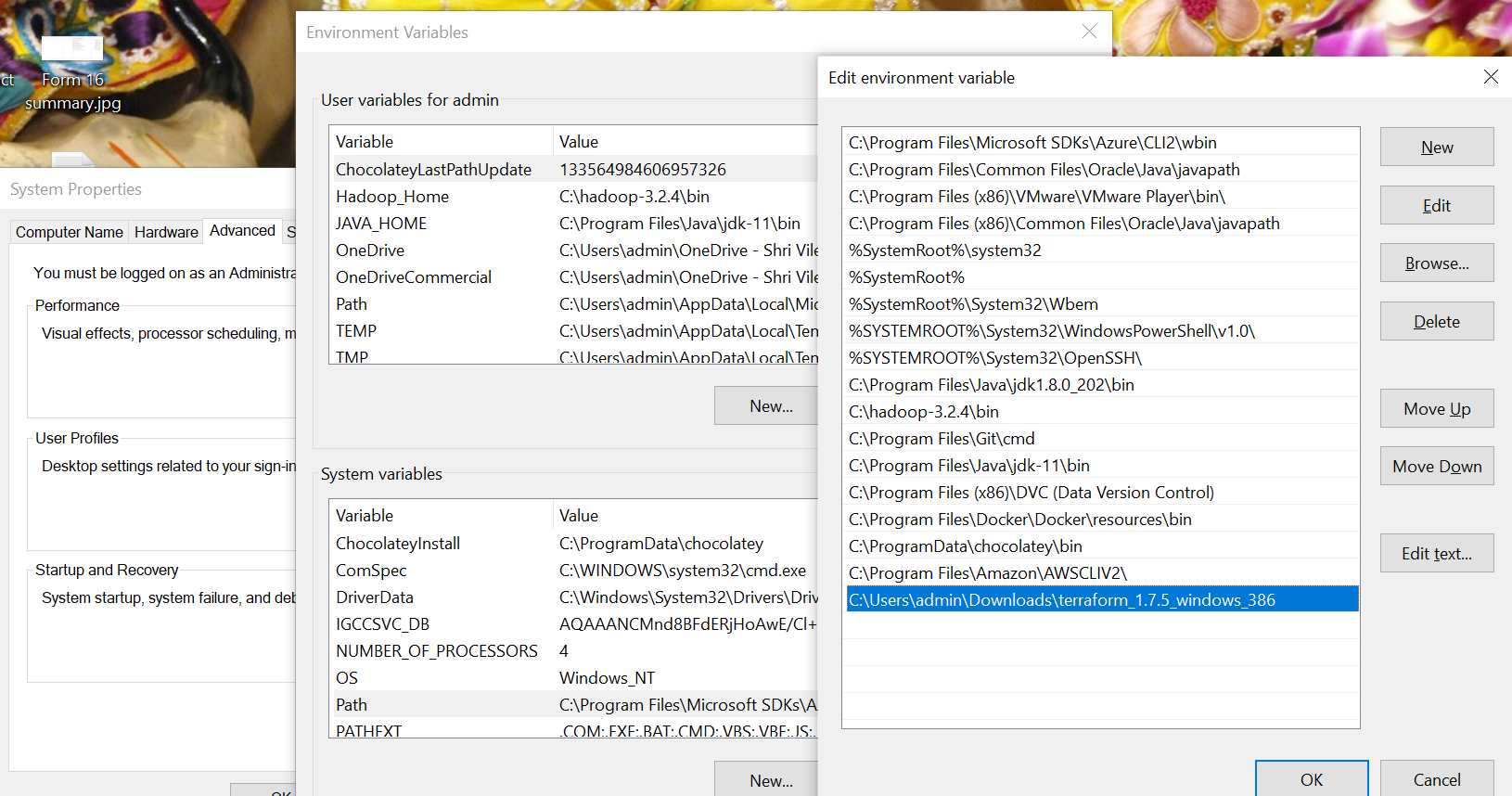
**Step1: install terraform: download the package for terraform from the given link**

**https://developer.hashicorp.com/terraform/install**



**Step2: setup environment variable path for terraform package**

**C:\Users\admin\Downloads\terraform\_1.11.3\_windows\_386**



**Step 3 :Check terraform install in your system with version of terraform**

C:\Users\djsce.student>terraform -version

Terraform v1.7.5

on windows\_386



**Step 4:**

Run docker desktop

**Step 5: Create folder terra🡪 module.tf # inside of the “module.tf “write following code**

terraform {

required\_providers {

docker = {

source = "kreuzwerker/docker"

version = "~> 2.23.0"

}

}

}

provider "docker" {

host = "npipe:////.//pipe//docker\_engine"

}

resource "docker\_image" "nginx" {

name = "nginx"

keep\_locally = false

}

resource "docker\_container" "nginx" {

image = docker\_image.nginx.image\_id

name = "tutorial"

ports {

internal = 80

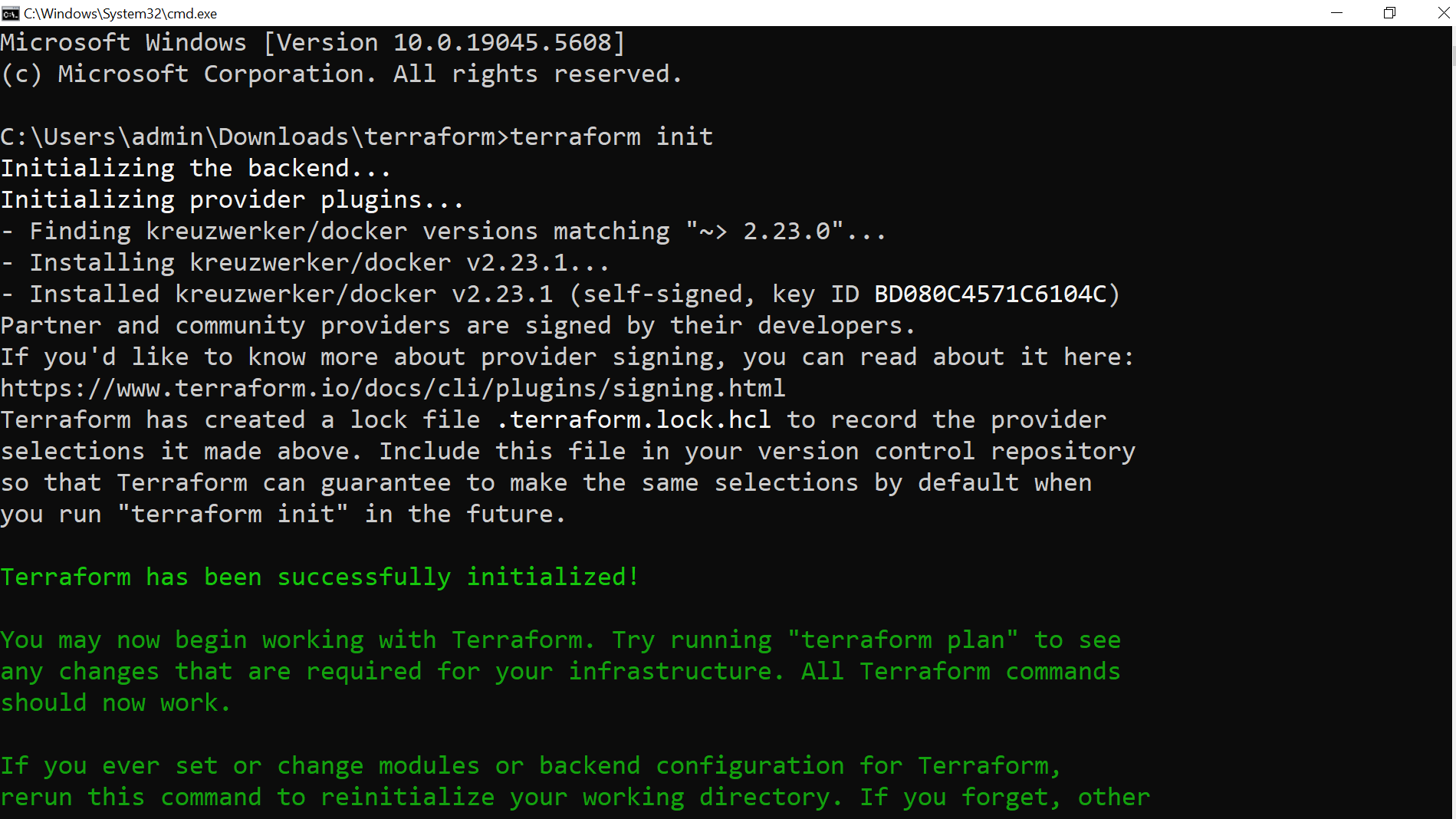
external = 8000

}

}

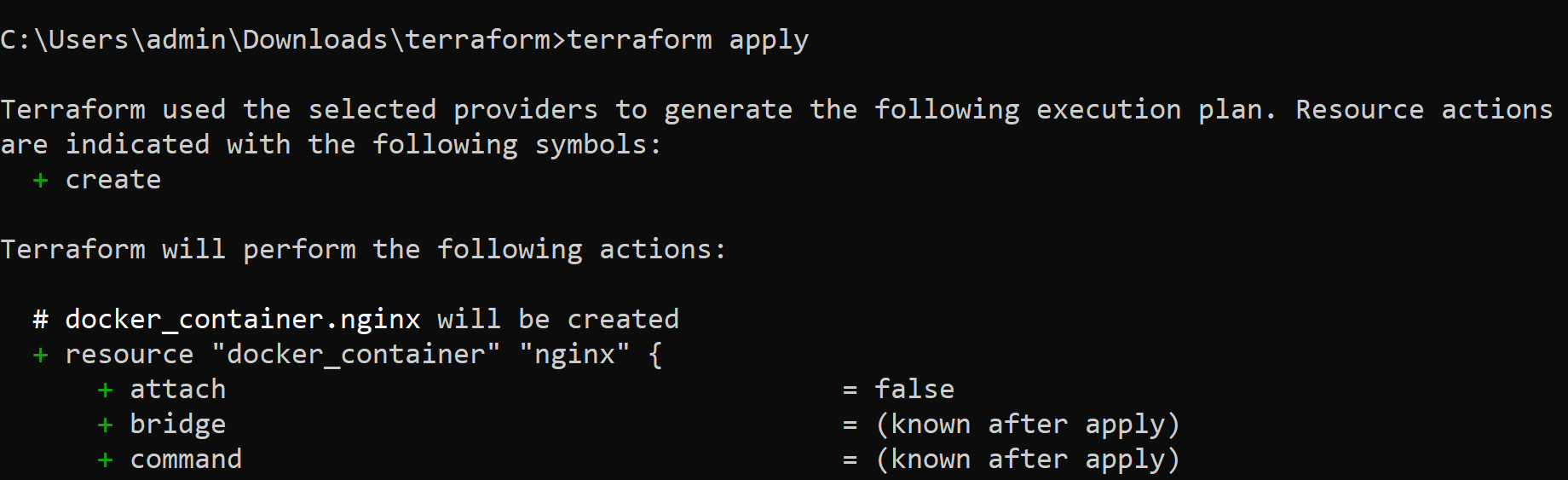
**Step7: Initialize the project, which downloads a plugin called a provider that lets Terraform interact with Docker.**

**terraform init**

****

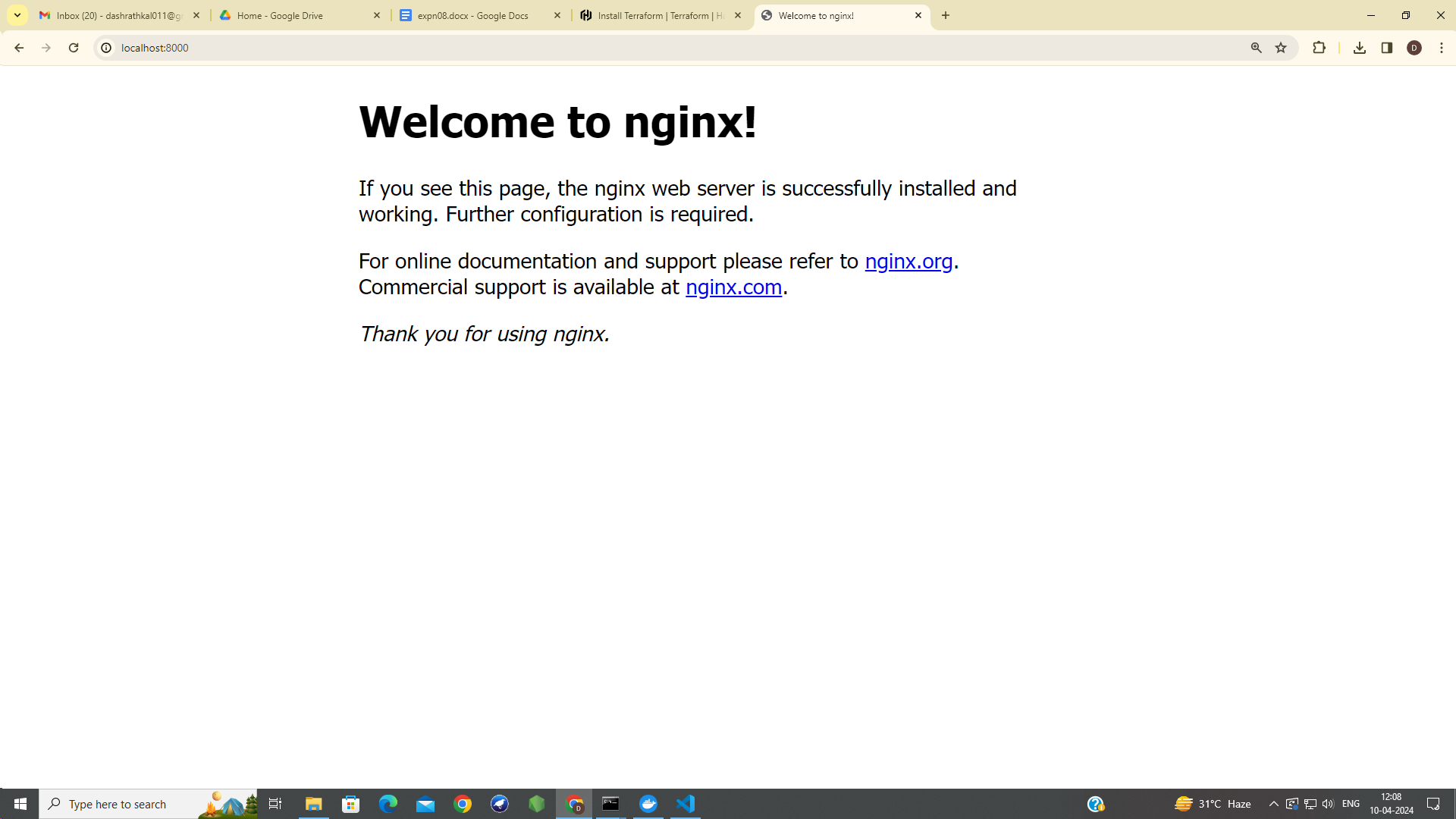
**Step 9: Provision the NGINX server container with apply. When Terraform asks you to confirm type yes and press ENTER**

**terraform apply**

****

**Step10:**

**Verify the existence of the NGINX container by visiting**[**localhost:8000**](http://localhost:8000/)**in your web browser or running docker ps to see the container.**



**Step 11: Verify the existence of the NGINX container by visiting**[**localhost:8000**](http://localhost:8000/)**in your web browser or running docker ps to see the container.**

**Step 12: This command is to see the container.**

**docker ps**

**Second Script**

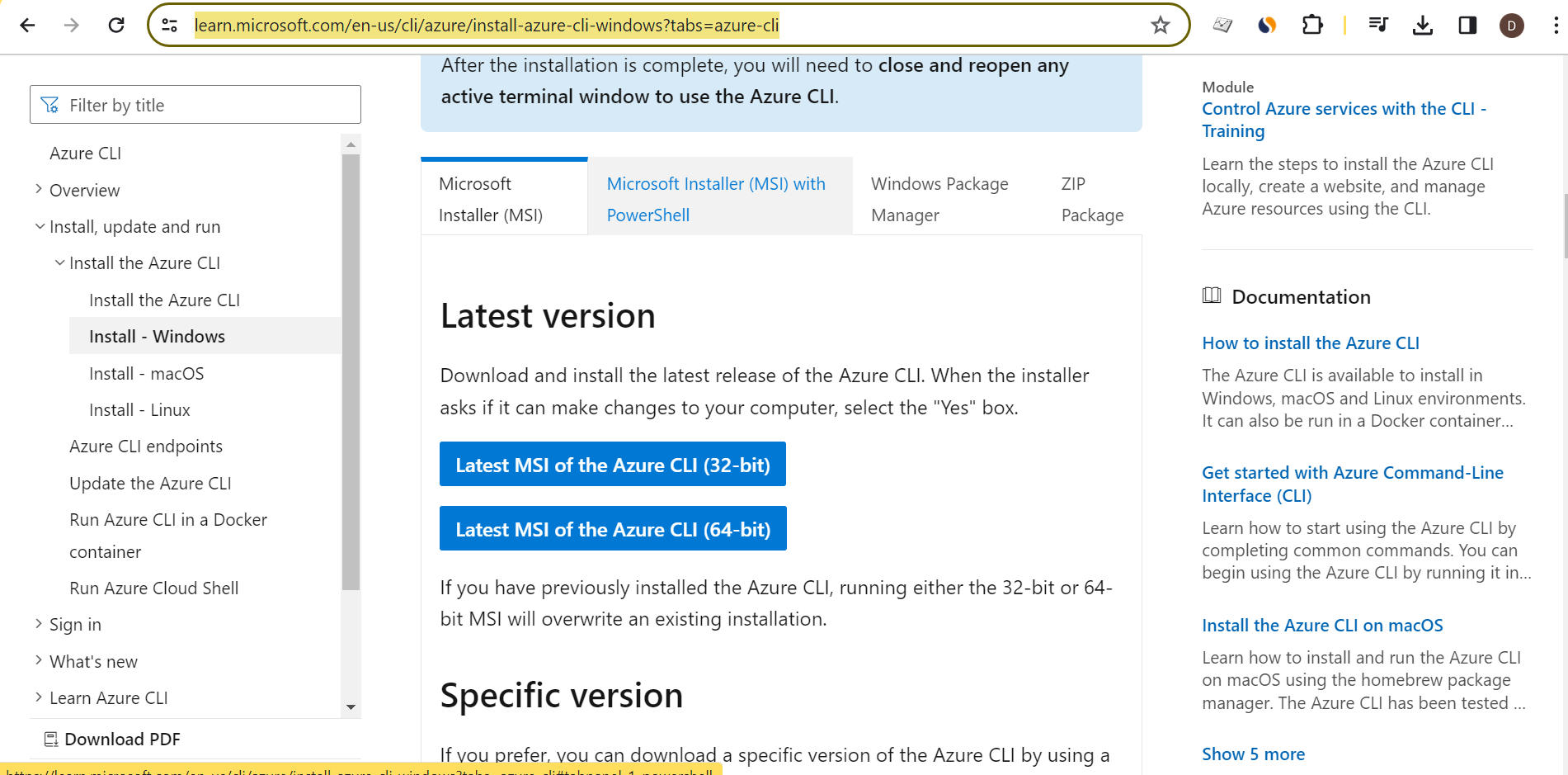
**Terraform deploying resources on Azure cloud**

**Step1: install Azure Cli**

**Link:** <https://developer.hashicorp.com/terraform/tutorials/azure-get-started/azure-build>

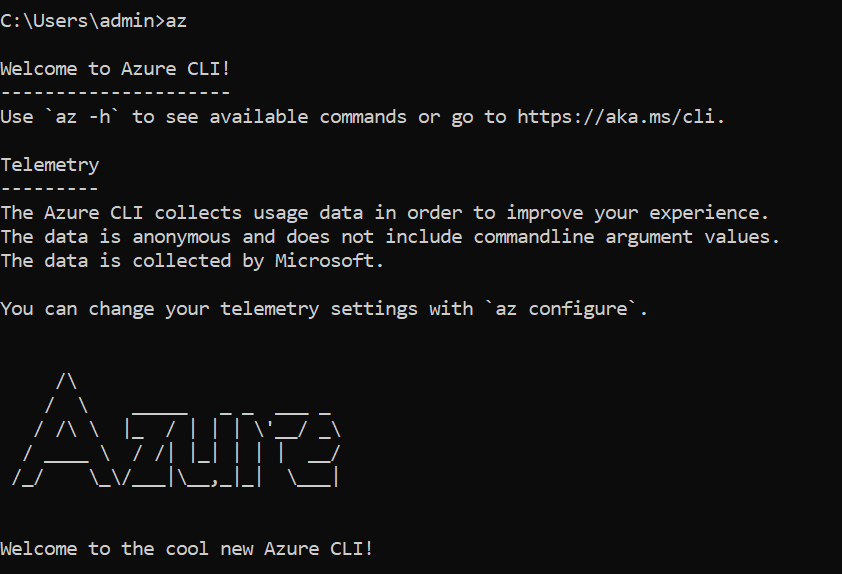
**WinGet (Windows Package Manager)**

winget install --exact --id Microsoft.AzureCLI

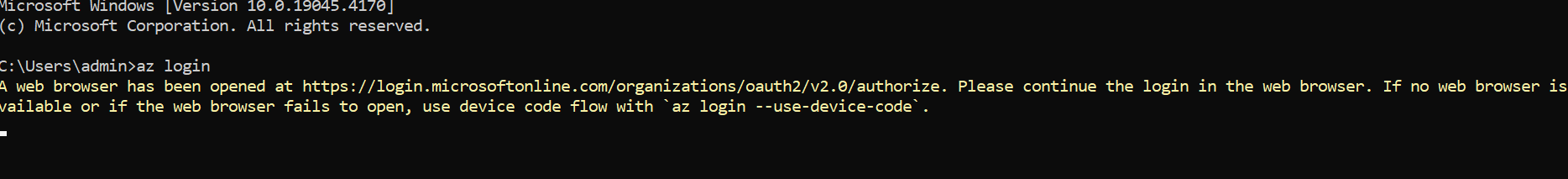


## **Step 2: Run the Azure CLI**

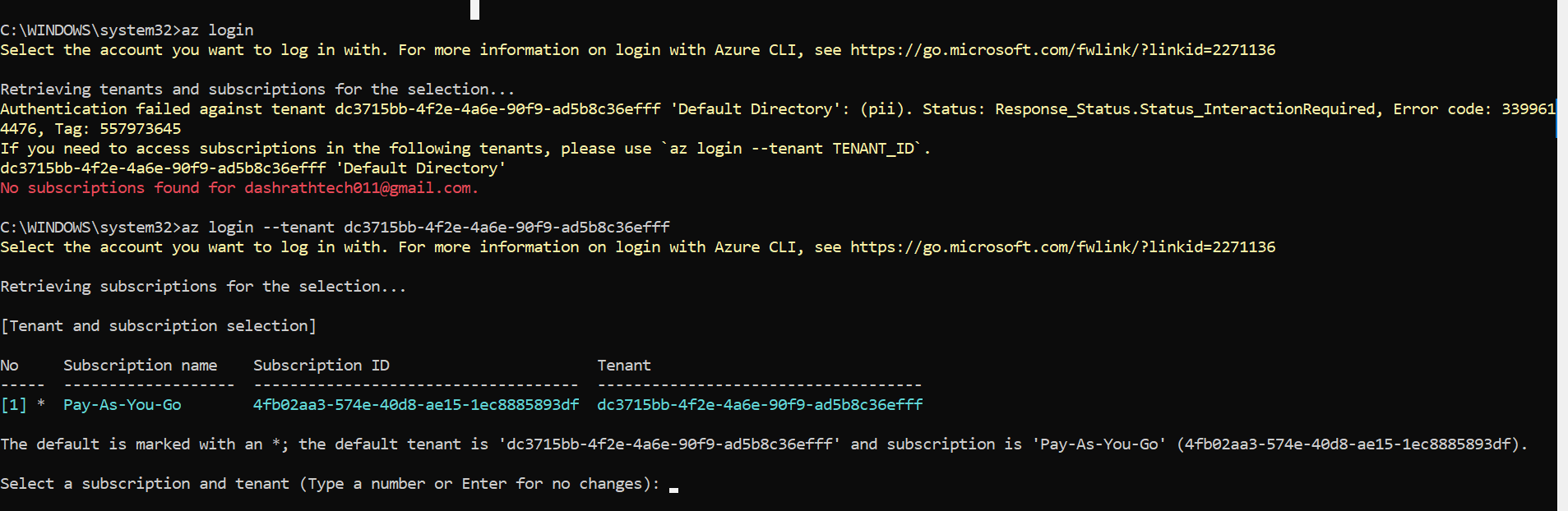
After installation, close and reopen any active terminal window. Run the Azure CLI with the **az** command from either Windows Command Prompt or PowerShell.



**Step 3: In your terminal, use the Azure CLI tool to setup your account permissions for azure cloud**



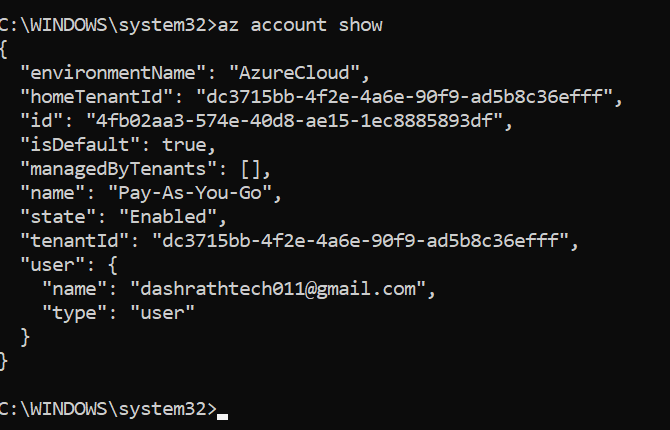
Step 4: Your browser will open and prompt you to enter your Azure login credentials. After successful authentication, your terminal will display your subscription information.



az login --tenant dc3715bb-4f2e-4a6e-90f9-ad5b8c36efff # give ur own account id

Select a subscription and tenant (Type a number or Enter for no changes): #enter key

**step 4: C:\WINDOWS\system32>az account show**



Step 5: Create Folder terraform2🡪 main.tf write code for resource provisioning as below

# Configure the Azure provider

terraform {

  required\_providers {

    azurerm = {

      source  = "hashicorp/azurerm"

      version = "~> 2.23.0"

    }

  }

  required\_version = ">= 1.1.0"

}

provider "azurerm" {

  features {}

}

resource "azurerm\_resource\_group" "rg" {

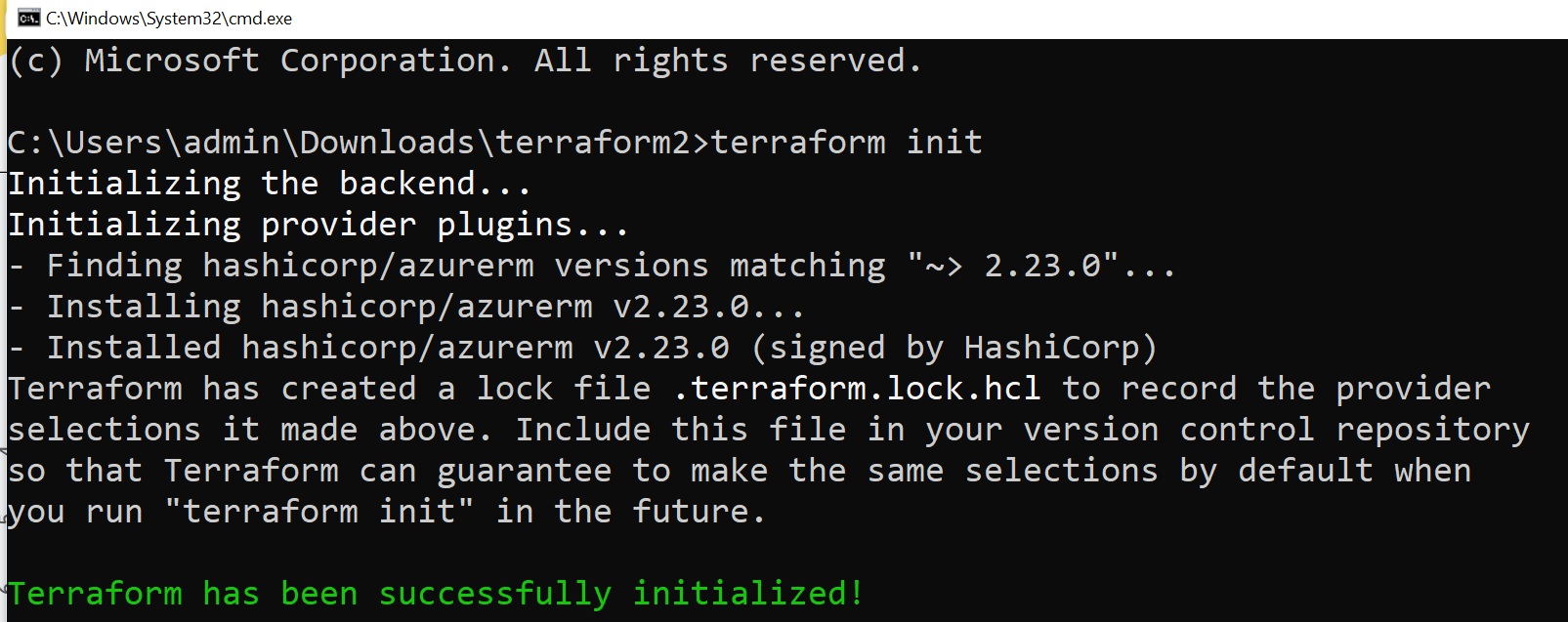
  name     = "VESP"

  location = "westus2"

}

**Step5:**

**PS C:\Users\admin\Documents\AI\exp8> terraform init**



**Step6:**

**PS C:\Users\admin\Documents\AI\exp8> terraform plan**

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

# azurerm\_resource\_group.rg will be created

+ resource "azurerm\_resource\_group" "rg" {

+ id = (known after apply)

+ location = "westus2"

+ name = "myTFResourceGroup"

}

Plan: 1 to add, 0 to change, 0 to destroy.

──────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────────

Note: You didn't use the -out option to save this plan, so Terraform can't guarantee to take exactly these actions if you run "terraform

apply" now.

**Step7:**

**PS C:\Users\admin\Documents\AI\exp8> terraform apply**

Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:

+ create

Terraform will perform the following actions:

# azurerm\_resource\_group.rg will be created

+ resource "azurerm\_resource\_group" "rg" {

+ id = (known after apply)

+ location = "westus2"

+ name = "myTFResourceGroup"

}

Plan: 1 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?

Terraform will perform the actions described above.

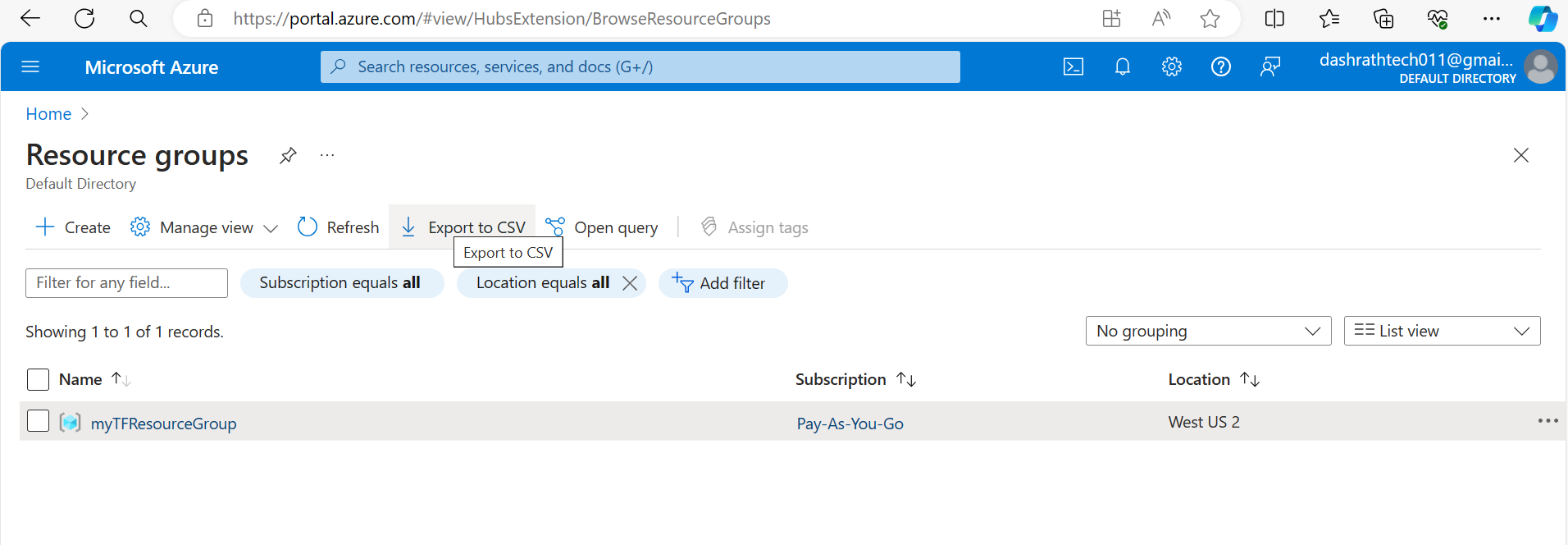
Only 'yes' will be accepted to approve.

Enter a value: yes

azurerm\_resource\_group.rg: Creating...

azurerm\_resource\_group.rg: Creation complete after 5s [id=/subscriptions/4fb02aa3-574e-40d8-ae15-1ec8885893df/resourceGroups/myTFResourceGroup]

Apply complete! Resources: 1 added, 0 changed, 0 destroyed.



Conclusion:

References:

<https://www.youtube.com/watch?v=knAAakyRbiE&t=637s>

<https://www.youtube.com/watch?v=FUa9TOd7LF0>