**Lab Manual- Manage Azure Stoarge Infrastcrture Using Terraform**

Table of Contents

[1 OBJECTIVE 3](#_Toc183864889)

[2 What is Terraform 3](#_Toc183864890)

[3 Terraform Provider 4](#_Toc183864891)

[3.1 Key Concepts of a Terraform Provider: 5](#_Toc183864892)

[3.1.1 **Purpose:** 5](#_Toc183864893)

[3.1.2 **Examples:** 5](#_Toc183864894)

[3.1.3 **Configuration:** 5](#_Toc183864895)

[3.1.4 **Authentication:** 5](#_Toc183864896)

[3.1.5 **Resource Types:** 5](#_Toc183864897)

[3.1.6 **Versioning:** 6](#_Toc183864898)

[3.1.7 **Terraform Registry:** 6](#_Toc183864899)

[3.1.8 **Plugin-Based Architecture:** 6](#_Toc183864900)

[3.1.9 **Benefits:** 6](#_Toc183864901)

[4 PRE-REQUISISTE 7](#_Toc183864902)

[5 Create Resource Group with Terraform Direct Definition approach 7](#_Toc183864903)

[5.1 Explanation: 8](#_Toc183864904)

[5.2 When to Use Direct Definition: 8](#_Toc183864905)

[5.3 Advantages: 8](#_Toc183864906)

[5.4 Disadvantages: 9](#_Toc183864907)

[5.5 Execute Terraform Script 9](#_Toc183864908)

[6 Create Resource Group with Terraform Variable File 11](#_Toc183864909)

[9 Create Resource Group with Terraform Runtime Input 17](#_Toc183864910)

[10 Update Resource Group with Tag 19](#_Toc183864911)

[11 Delete Resource Group 22](#_Toc183864912)

# OBJECTIVE

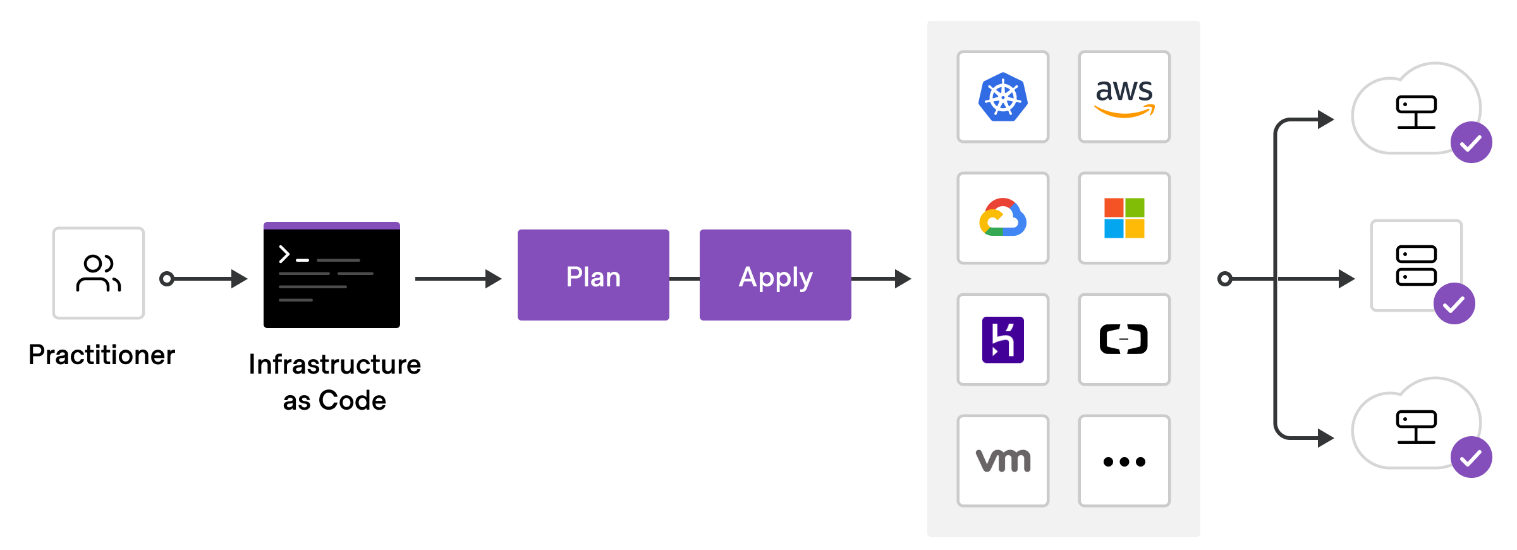
Terraform Azure providers enable you to manage all of your Azure infrastructure using the same declarative syntax and tooling. Using these providers you can:

* Provision core platform capabilities such as management groups, policies, users, groups, and policies. For more information, see [Terraform implementation of Cloud Adoption Framework Enterprise-scale](https://github.com/Azure/terraform-azurerm-caf-enterprise-scale#readme).
* Provision Azure DevOps Projects and pipelines to automate regular infrastructure and application deployments.
* Provision Azure resources required by your applications.

# What is Terraform

Terraform is HashiCorp's infrastructure as code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files, and manages your infrastructure's lifecycle. Using Terraform has several advantages over manually managing your infrastructure:

* Terraform can manage infrastructure on multiple cloud platforms.
* The human-readable configuration language helps you write infrastructure code quickly.
* Terraform's state allows you to track resource changes throughout your deployments.
* You can commit your configurations to version control to safely collaborate on infrastructure.
* To deploy infrastructure with Terraform:
  + **Scope** - Identify the infrastructure for your project.
  + **Author** - Write the configuration for your infrastructure.
  + **Initialize** - Install the plugins Terraform needs to manage the infrastructure.
  + **Plan** - Preview the changes Terraform will make to match your configuration.
  + **Apply** - Make the planned changes.



# Terraform Provider

* A **Terraform provider** is a plugin in Terraform that enables interaction with external APIs and services. It acts as a bridge between Terraform and the underlying infrastructure, platform, or service you want to manage. Providers allow Terraform to understand the available resources, their configurations, and how to interact with them to create, update, or delete resources.

## Key Concepts of a Terraform Provider:

### **Purpose:**

* + Providers manage the lifecycle of resources within the target service or platform.
  + They translate Terraform configuration into API calls to the target service.

### **Examples:**

* + **Cloud Providers:** aws, azurerm, google (for AWS, Azure, and Google Cloud respectively)
  + **Other Services:** kubernetes, docker, vault, random, etc.
  + **Custom Providers:** You can create custom providers for proprietary systems or unsupported APIs.

### **Configuration:**

* + Providers are specified in the Terraform configuration using the provider block.
  + For example, configuring the azurerm provider:

hcl

Copy code

provider "azurerm" {

features {}

subscription\_id = "your-subscription-id"

tenant\_id = "your-tenant-id"

client\_id = "your-client-id"

client\_secret = "your-client-secret"

}

### **Authentication:**

* + Providers often require authentication to interact with their APIs. This might involve API keys, environment variables, or service principals.

### **Resource Types:**

* + Each provider offers specific **resource types** and **data sources** that you can use in your Terraform configurations.
  + Example (Azure resource group):

hcl

Copy code

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

name = "example-resource-group"

location = "East US"

}

### **Versioning:**

* + Providers are versioned, and you can specify which version to use to ensure stability.
  + Example:

hcl

Copy code

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "~> 3.0"

}

}

}

### **Terraform Registry:**

* + Terraform providers are hosted on the Terraform Registry. Here, you can explore providers, their documentation, and usage examples.

<https://registry.terraform.io/browse/providers>

### **Plugin-Based Architecture:**

* + Providers are plugins, and Terraform downloads them when you run terraform init.

### **Benefits:**

* Simplifies infrastructure management by abstracting API interactions.
* Offers consistency across different platforms and services.
* Allows for infrastructure as code (IaC) to manage resources declaratively.
* If you are working with Terraform in an environment such as Azure, AWS, or on-premises solutions, the provider is the key component that enables interaction with those platforms.

# PRE-REQUISISTE

* Accounts in Azure
* A local Computer with 4 CPU, 16 GB RAM, 200 GB disk space
* An Azure tenant and access to a subscription, like **Owner** or **Contributor** rights.
* VS Code or other IDE. However, VS Code has a [Terraform extension](https://docs.microsoft.com/en-us/azure/developer/terraform/configure-vs-code-extension-for-terraform) to improve the authoring process.
* Terraform open-source command-line interface
* Azure CLI ([download](https://docs.microsoft.com/en-us/cli/azure/install-azure-cli)). This tutorial uses version 2.32.0.

# Create Resource Group with Terraform Direct Definition approach

The **Direct Definition** approach in Terraform refers to hardcoding values directly into the configuration files, rather than using variables, inputs, or external files to define them. This approach is simple and quick, making it useful for fixed, unchanging configurations that don’t need to be reused or customized oft

Consider this example where we directly define the values of the **Resource Group Name** and **Location** in the main.tf file:

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm" # Specifies the source for the Azure provider plugin.

version = "~>2.0" # Ensures the Azure provider plugin version is 2.x.x (compatible with this configuration).

}

}

}

provider "azurerm" {

features {} # Required for newer versions of the Azure provider, initializes provider features.

subscription\_id = "49c56ee8-d443-4854-a62c-3a0aae84ac6f" # Your Azure subscription ID.

tenant\_id = "be04fbd5-6b00-412c-a86c-ca105b5cce90" # Your Azure Active Directory (AAD) tenant ID.

client\_id = "0b381472-3197-49d4-a324-f1a96a23c8a7" # The client ID of the service principal for authentication.

client\_secret = "PiX8Q~CqWaqiC4Bh~rALxAl56VFmkllYgNJmHc-D" # The client secret of the service principal.

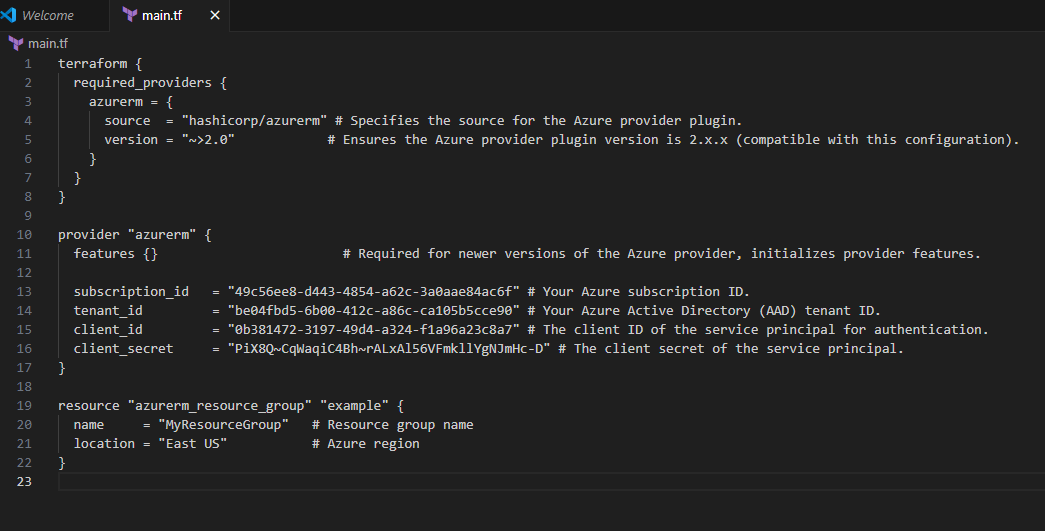
}

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

name = "MyResourceGroup" # Resource group name

location = "East US" # Azure region

}



## Explanation:

* **Direct Definition**: The values for name and location are directly written in the configuration file.
* **No Variables Used**: There are no variables or external files used to manage these values.
* **Hardcoded Values**: These are fixed values, meaning the resource group name is always MyFixedResourceGroup, and the location is always East US. You can’t easily change them without modifying the configuration file.

## When to Use Direct Definition:

* **Fixed Environments**: When you know the exact configuration, and it is unlikely to change across different environments or use cases (e.g., a demo environment or a test setup).
* **Quick Prototyping**: If you're quickly testing something or don’t need to make the configuration flexible or reusable.
* **Simplicity**: Ideal for small, simple configurations where there's no need to modify parameters or reuse the configuration in different environments.

## Advantages:

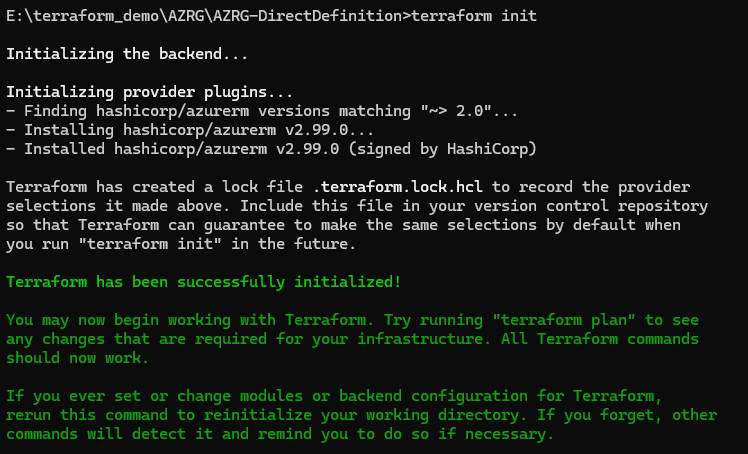
1. **Simplicity**: No need to deal with variables, making it easy to understand and use.
2. **Speed**: For simple, one-off projects, directly defining values saves time as you don’t need to set up variables or external files.
3. **No External Dependencies**: Everything is contained within the main.tf file, making it self-sufficient.

## Disadvantages:

1. **Lack of Flexibility**: Since values are hardcoded, changing the configuration for different environments (e.g., dev, staging, production) requires manual edits in the Terraform file.
2. **Maintenance**: As your infrastructure grows, maintaining multiple hardcoded values in large configurations becomes cumbersome and error-prone.
3. **Reuse Limitations**: If you need to reuse this configuration in other contexts (e.g., for different regions or teams), you'll have to copy and modify the configuration manually.

## Execute Terraform Script

**terraform init**

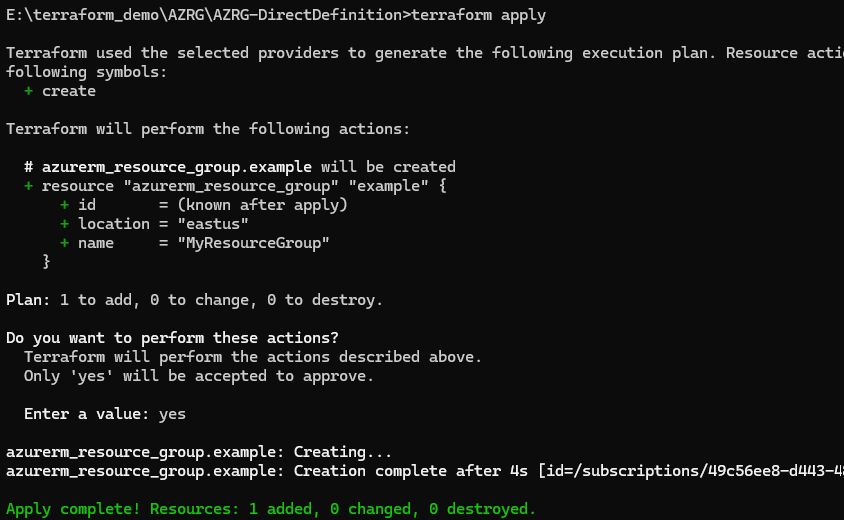


**terraform plan**

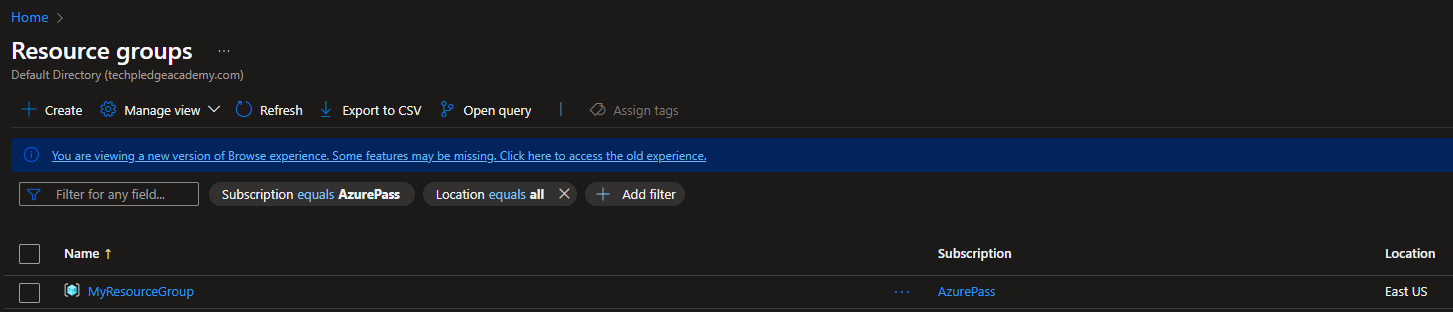
A screenshot of a computer program

Description automatically generated

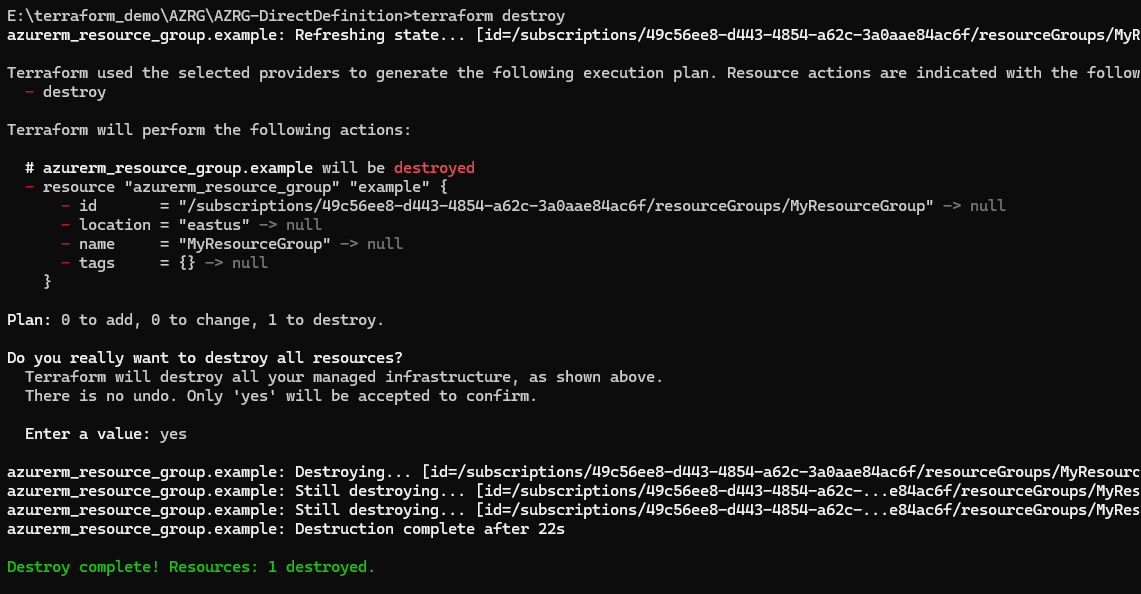
**terraform apply**



Verify on Portal



**terraform destroy**



# Create Resource Group with Terraform Variable File

**Data Type**

|  |  |
| --- | --- |
| **Type** | **Example** |
| string | "East US" |
| number | 3 |
| bool | true / false |
| list | ["East US", "West Europe"] |
| map | { dev = "East US", prod = "West US" } |
| object | { name = string, size = number } |
| tuple | ["East US", 3, true] |

**In Terraform, variables are mainly of 3 types:**

| **Type** | **Purpose** | **Example** |
| --- | --- | --- |
| **Input Variables (variable)** | Values you pass *into* the configuration (like user input) | variable "region" {} |
| **Local Values (locals)** | *Temporary* variables inside your code (only in the same module/file) | locals { rg\_name = "demoRG" } |
| **Output Values (output)** | Values you want to *show or share* after apply | output "rg\_name" {} |

**1️ Input Variables (variable)**

* Think: **Settings you want to customize when running.**
* Declared using variable block.

Example:

variable "location" {

description = "Azure region"

type = string

default = "East US"

}

➡️ You can override it using **terraform.tfvars**, command line, etc.

**2️ Local Variables (locals)**

* Think: **Shortcuts** inside your Terraform file.
* Good for reusing a value, doing simple calculations, cleaner code.
* Cannot be changed *outside* — purely internal.

Example:

locals {

rg\_name = "demo-resource-group"

}

➡️ You use it like:

**name = local.rg\_name**

✅ **Locals** make the code DRY (Don't Repeat Yourself).

**3️ Output Variables (output)**

* Think: **Results you want to display after terraform apply.**
* Useful for showing IP addresses, resource names, etc.

Example:

output "resource\_group\_name" {

value = azurerm\_resource\_group.example.name

}

➡️ After apply, Terraform will print that value on the screen.

* A **Variable File** in Terraform is a powerful tool to separate configuration values from the resource definitions. It helps:
* **Increase modularity** by keeping values and logic separate.
* **Improve reusability** across different environments.
* **Allow easier maintenance** of values in larger, more complex infrastructures.
* By using variable files, you can manage dynamic configurations and avoid hardcoding values directly into your main Terraform files, making your infrastructure code cleaner and more scalable.

1. Create Another Folder and Create **Main.tf, Variable.tf and tfvars** file inside it.

**# main.tf**

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm" # Specifies the source for the Azure provider plugin.

version = "~>2.0" # Ensures the Azure provider plugin version is 2.x.x (compatible with this configuration).

}

}

}

provider "azurerm" {

features {} # Required for newer versions of the Azure provider, initializes provider features.

subscription\_id = "49c56ee8-d443-4854-a62c-3a0aae84ac6f" # Your Azure subscription ID.

tenant\_id = "be04fbd5-6b00-412c-a86c-ca105b5cce90" # Your Azure Active Directory tenant ID.

client\_id = "0b381472-3197-49d4-a324-f1a96a23c8a7" # The client ID of the service principal for authentication.

client\_secret = "PiX8Q~CqWaqiC4Bh~rALxAl56VFmkllYgNJmHc-D" # The client secret of the service principal.

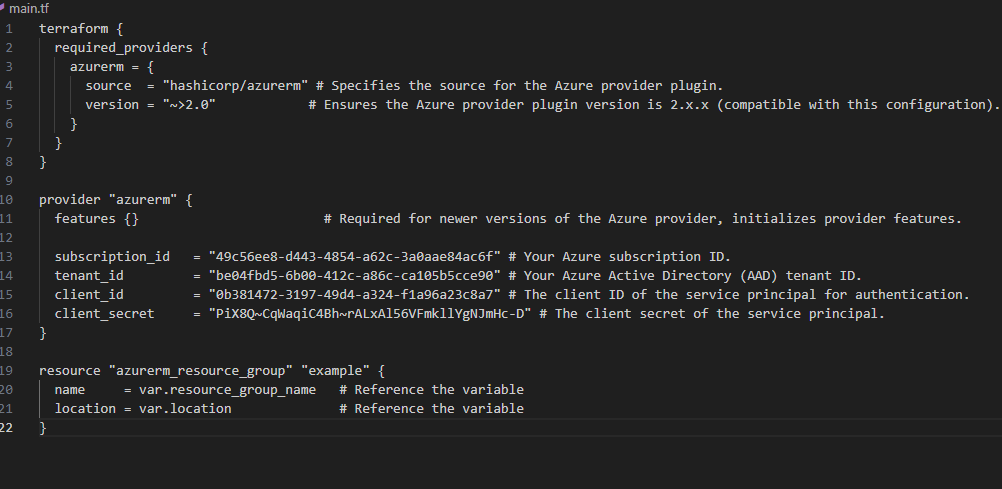
}

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

name = var.resource\_group\_name # Reference the variable

location = var.location # Reference the variable

}



**# variables.tf**

variable "resource\_group\_name" {

description = "The name of the Resource Group"

type = string

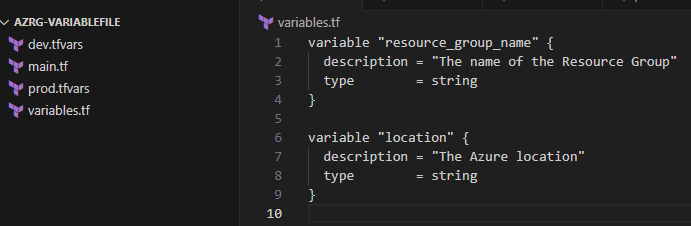
}

variable "location" {

description = "The Azure location"

type = string

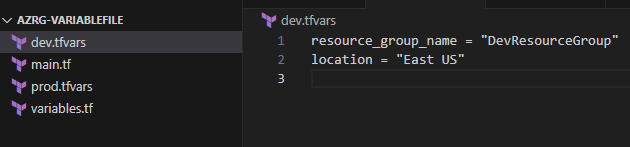
}



**# dev.tfvars**

resource\_group\_name = "DevResourceGroup"

location = "East US"



**# prod.tfvars**

resource\_group\_name = "ProdResourceGroup"

location = "West US"

A screenshot of a computer

Description automatically generated

1. **Example Overview**
2. **Main Configuration (main.tf):**
   * **This is where your resources are defined. Variables are referenced but not given values directly.**
3. **Variable Definitions (variables.tf):**
   * **This file contains the definition of each variable that you use in your configuration. It specifies what type of value the variable should be (e.g., string, number, list).**
4. **Variable Values (terraform.tfvars):**
   * **This file contains the actual values that are assigned to the variables defined in variables.tf. When you run terraform apply, Terraform loads these values and uses them in the configuration.**
5. **Advantages of Using Variable Files**
   * **Separation of Concerns: The logic for defining resources is separate from the values you use for those resources. This makes the configuration more modular and maintainable.**
   * **Reusability: You can create different .tfvars files for different environments (e.g., dev.tfvars, prod.tfvars) to easily switch configurations between environments.**
   * **Improved Security: By using variable files, sensitive data (like passwords or API keys) can be passed in securely, potentially avoiding hardcoding sensitive values in the main configuration.**
   * **Flexibility: Allows dynamic configuration where values can be customized without modifying the Terraform configuration files directly.**
6. Run [terraform init](https://www.terraform.io/docs/commands/init.html) to initialize the Terraform deployment. This command downloads the Azure modules required to manage your Azure resources.

**terraform init**

Text

Description automatically generated

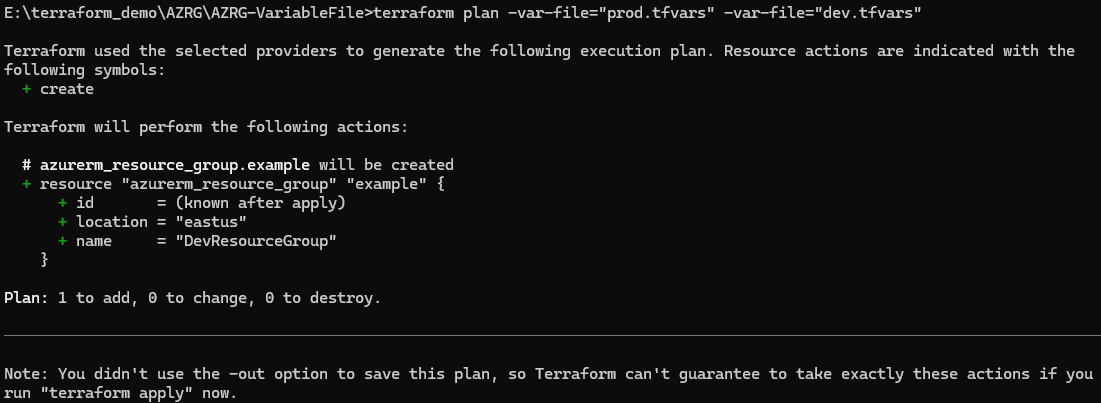
1. Run [terraform plan](https://www.terraform.io/docs/commands/plan.html) to create an execution plan,

**Order of .tfvars files**: Since prod.tfvars is listed second, its values override the ones from dev.tfvars.

If you wanted the values from dev.tfvars to take precedence, you would need to switch the order of the .tfvars files in the terraform plan command:

**terraform plan -var-file="dev.tfvars"**

**terraform plan -var-file="prod.tfvars" -var-file="dev.tfvars"**

****

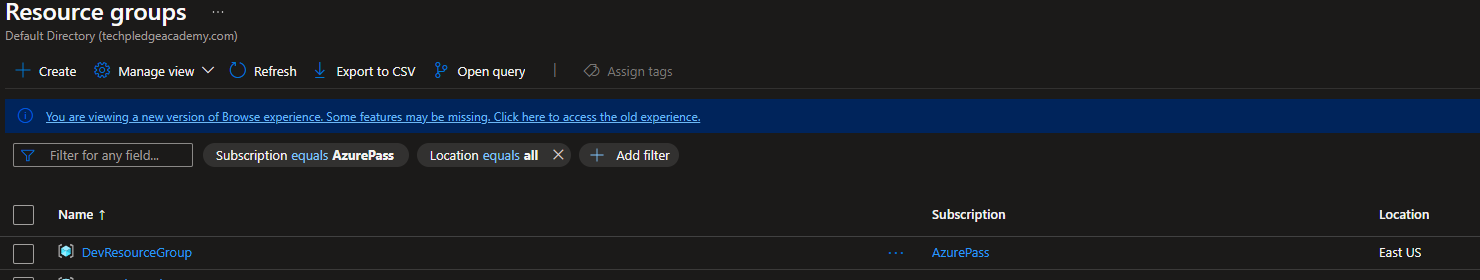
1. Run [terraform apply](https://www.terraform.io/docs/commands/apply.html) to apply the execution plan to your cloud infrastructure.

**terraform apply -var-file="dev.tfvars"**

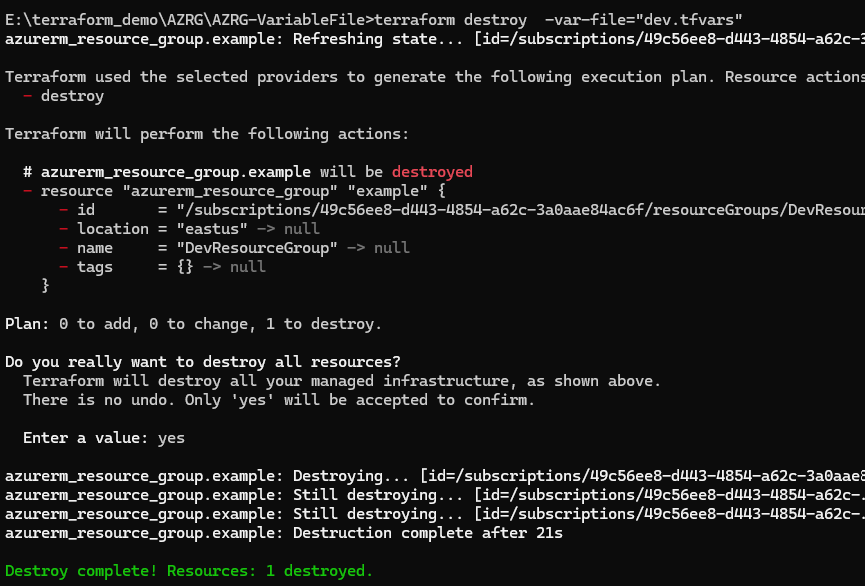
A screenshot of a computer program

Description automatically generated

1. Verify on Azure Portal

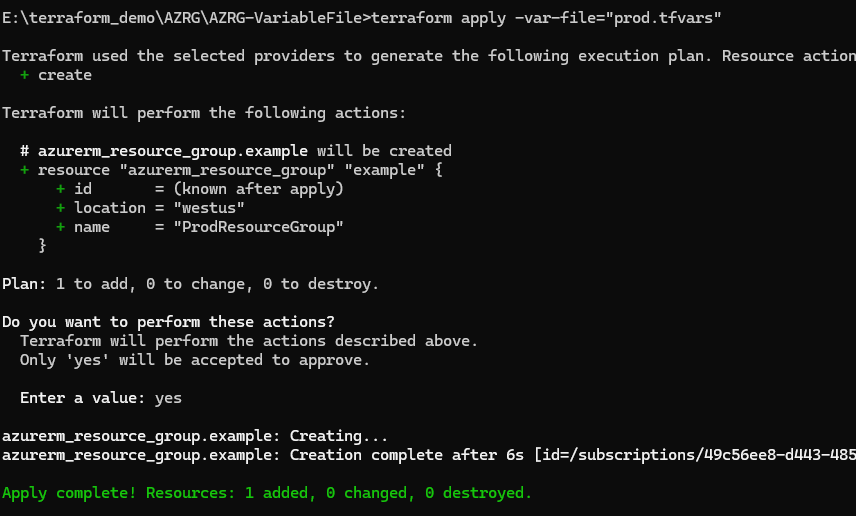


1. Delete the Resources

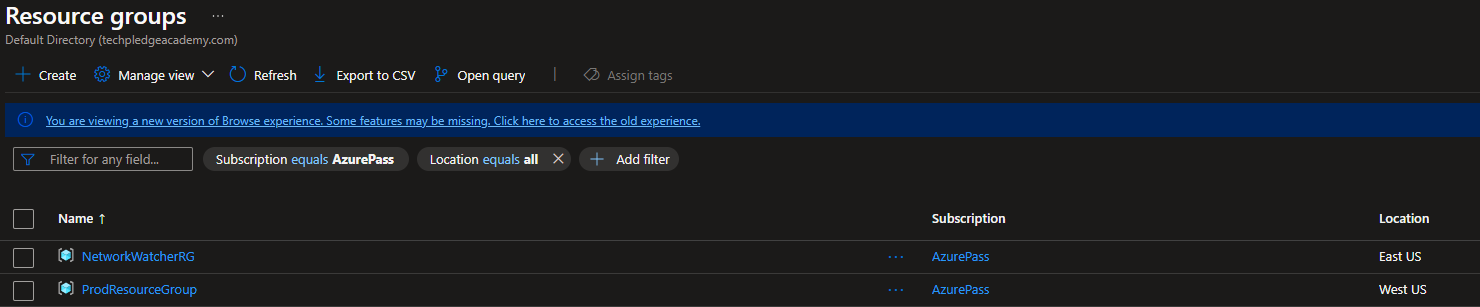


1. Run [terraform apply](https://www.terraform.io/docs/commands/apply.html) for prod.

**terraform apply -var-file="prod.tfvars"**



1. Verify on Azure Portal



1. Run [terraform](https://www.terraform.io/docs/commands/apply.html) destroy

**terraform destroy -var-file="dev.tfvars"**

A screenshot of a computer program

Description automatically generated

# Create Resource Group with Terraform Runtime Input

**Runtime Input** in Terraform refers to the ability to provide values for variables dynamically, typically at the time of running Terraform commands (terraform plan or terraform apply). This provides more flexibility compared to defining static values in .tfvars files or hardcoding them directly into the Terraform configuration. You can supply values interactively, through environment variables, or in other ways when Terraform is executed, making your deployments more adaptable to different environments and use cases.

**Main.tf**

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm" # Specifies the source for the Azure provider plugin.

version = "~>2.0" # Ensures the Azure provider plugin version is 2.x.x (compatible with this configuration).

}

}

}

provider "azurerm" {

features {} # Required for newer versions of the Azure provider, initializes provider features.

subscription\_id = "49c56ee8-d443-4854-a62c-3a0aae84ac6f" # Your Azure subscription ID.

tenant\_id = "be04fbd5-6b00-412c-a86c-ca105b5cce90" # Your Azure Active Directory (AAD) tenant ID.

client\_id = "0b381472-3197-49d4-a324-f1a96a23c8a7" # The client ID of the service principal for authentication.

client\_secret = "PiX8Q~CqWaqiC4Bh~rALxAl56VFmkllYgNJmHc-D" # The client secret of the service principal.

}

variable "RGName" {

description = "The name of the resource group" # Provides details about what this variable represents.

type = string # Specifies the variable type as a string.

}

variable "location" {

description = "The Azure region where the resource group will be created" # Explains that this variable specifies the Azure region.

type = string # Defines the type as string.

}

variable "tags" {

description = "Tags to be applied to the resource group" # Provides additional metadata for the resource group.

type = map(string) # Specifies the variable type as a map of strings.

default = { # Sets default key-value pairs for tags.

Environment = "Development"

//Project = "TerraformDemo"

}

}

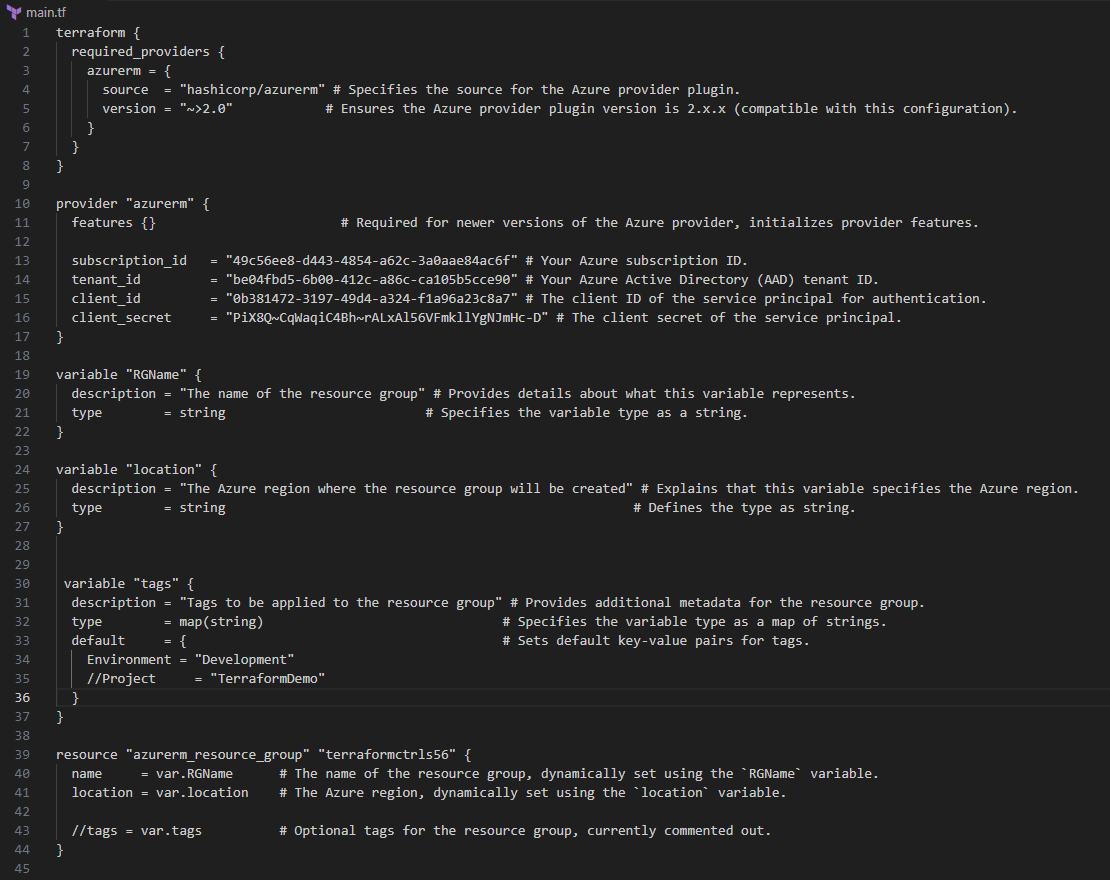
resource "azurerm\_resource\_group" "terraformctrls56" {

name = var.RGName # The name of the resource group, dynamically set using the `RGName` variable.

location = var.location # The Azure region, dynamically set using the `location` variable.

//tags = var.tags # Optional tags for the resource group, currently commented out.

}



* **Runtime Input** provides flexibility in dynamic deployments by allowing you to supply values for Terraform variables at the time of running terraform plan or terraform apply. This allows for:
* **Interactive prompts** for input during execution.
* **Environment variables** to pass values securely.
* **Command-line flags** to override values without changing configuration files.
* **Integration with external systems** for dynamic input.
* This dynamic input capability helps create more adaptable and reusable Terraform configurations for a wide range of environments and use cases.

Run

Terraform init

Terraform plan – when ask type any resource group name and location

Terraform apply – when ask type anyresource group name and location

# Update Resource Group with Tag

1. Update the Main.tf created for Resource Group AS highlighted Below. **Uncomment** the **Tag Part**

explain each line as comment : terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "~>2.0"

}

}

}

provider "azurerm" {

features {}

subscription\_id = "49c56ee8-d443-4854-a62c-3a0aae84ac6f"

tenant\_id = "be04fbd5-6b00-412c-a86c-ca105b5cce90"

client\_id = "0b381472-3197-49d4-a324-f1a96a23c8a7"

client\_secret = "PiX8Q~CqWaqiC4Bh~rALxAl56VFmkllYgNJmHc-D"

}

variable "RGName" {

description = "The name of the resource group"

type = string

}

variable "location" {

description = "The Azure region where the resource group will be created"

type = string

}

variable "tags" {

description = "Tags to be applied to the resource group"

type = map(string)

default = {

Environment = "Development"

Project = "TerraformDemo"

}

}

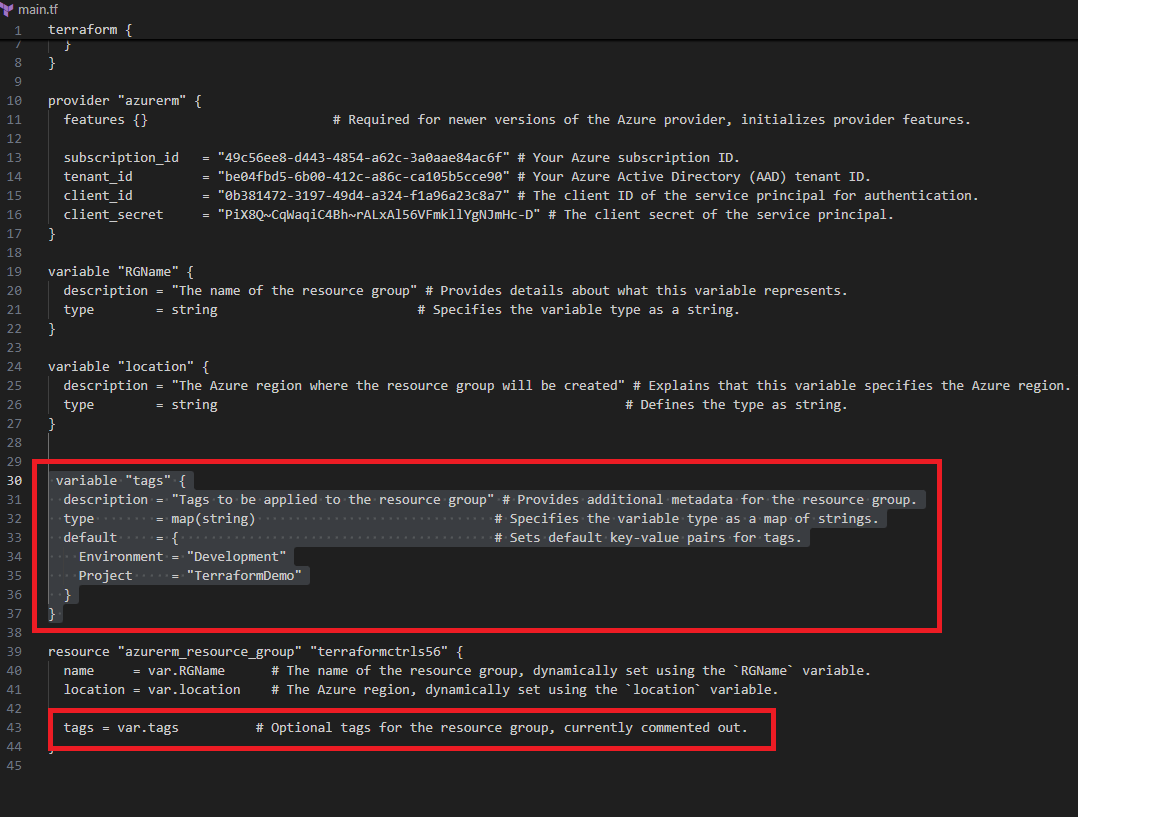
resource "azurerm\_resource\_group" "terraformctrls56" {

name = var.RGName

location = var.location

tags = var.tags

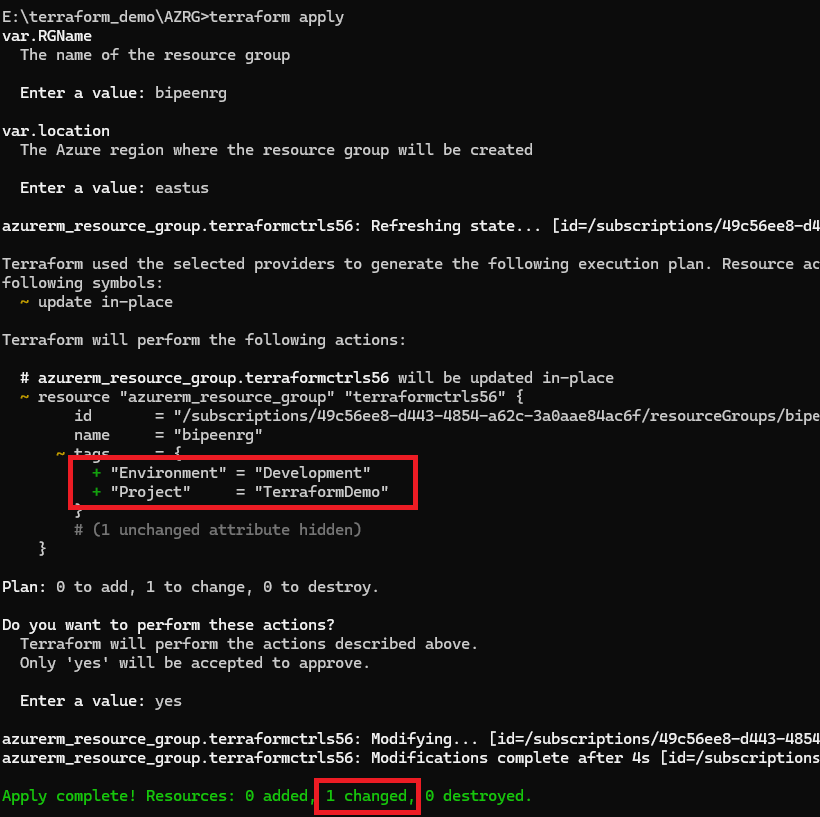
}



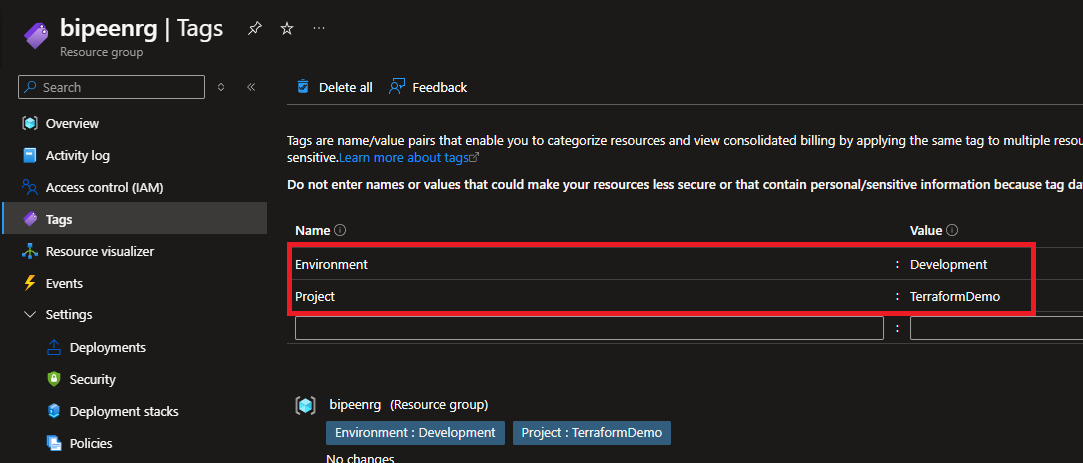
1. Run [terraform apply](https://www.terraform.io/docs/commands/apply.html) to apply the execution plan to your cloud infrastructure. when ask type **Resoure Group Name** and Location as **“EastUS“ .**

**terraform Apply**

**Note the difference it add new Tag**



1. Go to Azuree Portal and Verify



# Delete Resource Group

1. Run [terraform destroy](https://www.terraform.io/docs/commands/apply.html) to destroy cloud infrastructure created. when ask type **Resoure Group Name** and Location as **“EastUS“ .**

