**1. What is Terraform?**

Terraform is an **Infrastructure as Code (IaC)** tool developed by **HashiCorp** that allows you to define, provision, and manage infrastructure on **Azure** in a **declarative** way.

**Why Use Terraform with Azure?**

✅ Automates Azure infrastructure provisioning

✅ Manages state to track changes

✅ Works with multiple Azure services (VMs, Storage, AKS, Networking, etc.)

✅ Supports modular & reusable configurations

✅ Integrates with Azure DevOps for CI/CD automation

**2. Install Terraform for Azure**

**Step 1: Install Terraform**

* **Windows:**

choco install terraform

* **Linux & Mac:**

brew install terraform

**Step 2: Verify Installation**

terraform -v

**Step 3: Install Azure CLI**

Terraform uses **Azure CLI** for authentication.

* Install Azure CLI:

curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash

* Verify Azure CLI Installation:

az --version

* **Login to Azure**:

az login

**3. Terraform Basic Structure in Azure**

Terraform configurations are written in .tf files. A basic project structure looks like this:

azure-terraform/

│── main.tf # Main configuration (Resource creation)

│── variables.tf # Input variables

│── outputs.tf # Outputs after deployment

│── providers.tf # Provider configuration (Azure)

│── terraform.tfvars # Default variable values

│── backend.tf # Remote state storage

**4. Understanding Each Terraform File**

**1️⃣ providers.tf – Configuring Azure Provider**

This file configures **Terraform to use Azure**.

provider "azurerm" {

features {}

}

* azurerm is the provider for Azure resources.
* features {} is required for AzureRM provider.

**2️⃣ main.tf – Defining Resources**

This file contains **infrastructure resources**.

Example: Creating an **Azure Resource Group & Virtual Network**

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

name = "my-resource-group"

location = "East US"

}

resource "azurerm\_virtual\_network" "vnet" {

name = "my-vnet"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

address\_space = ["10.0.0.0/16"]

}

**3️⃣ variables.tf – Defining Variables**

Variables allow customization **without modifying code**.

variable "resource\_group\_name" {

description = "Azure Resource Group Name"

type = string

default = "my-resource-group"

}

variable "location" {

description = "Azure Region"

type = string

default = "East US"

}

**4️⃣ terraform.tfvars – Assigning Variable Values**

This file **stores default values** for variables.

resource\_group\_name = "prod-resource-group"

location = "West Europe"

**5️⃣ outputs.tf – Getting Outputs**

Outputs display important values **after execution**.

Example: Get the **Resource Group Name**

output "resource\_group\_name" {

value = azurerm\_resource\_group.rg.name

}

**6️⃣ backend.tf – Storing Terraform State in Azure**

Terraform **State** keeps track of your infrastructure.

To store it in **Azure Storage**, use:

terraform {

backend "azurerm" {

resource\_group\_name = "tf-state-rg"

storage\_account\_name = "tfstateaccount"

container\_name = "tfstate"

key = "terraform.tfstate"

}

}

📝 **Note:** You need to create an **Azure Storage Account** before using this.

**5. Terraform Workflow (Commands & Execution)**

**Step 1: Initialize Terraform**

terraform init

* Downloads required **AzureRM provider plugins**.
* Initializes Terraform **backend**.

**Step 2: Validate Configuration**

terraform validate

* Checks for syntax errors in **Terraform configuration**.

**Step 3: Plan Execution**

terraform plan

* Shows what changes Terraform **will apply**.
* **Dry-run** mode (doesn’t apply changes).

**Step 4: Apply Changes**

terraform apply

* Provisions resources in **Azure**.

✅ **Type** yes when prompted to confirm execution.

**Step 5: View Outputs**

terraform output

* Displays **resource values** from outputs.tf.

**Step 6: Destroy Infrastructure**

terraform destroy

* **Deletes all resources** provisioned.

**6. Writing a Complete Terraform Script for Azure**

**Example: Deploying an Azure Virtual Machine (VM)**

**Step 1: Install Terraform & Azure CLI**

Before you begin, make sure you have:

* Terraform installed: Download Terraform
* Azure CLI installed: Download Azure CLI
* Logged in to Azure: az login

**Step 2: Create a New Terraform Project**

1️⃣ **Create a project directory**

mkdir terraform-azure-vm && cd terraform-azure-vm

2️⃣ **Create Terraform configuration files** Inside this directory, create the following .tf files:

touch main.tf variables.tf outputs.tf provider.tf

**Step 3: Define the Azure Provider**

🔹 **File:** provider.tf

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "~>3.0"

}

}

}

provider "azurerm" {

features {}

}

✅ This configures Terraform to use **Azure Provider**.

**Step 4: Define the Variables**

🔹 **File:** variables.tf

variable "resource\_group\_name" {

description = "Azure Resource Group Name"

type = string

default = "myResourceGroup"

}

variable "location" {

description = "Azure Region"

type = string

default = "East US"

}

variable "vm\_name" {

description = "Name of the Virtual Machine"

type = string

default = "myVM"

}

variable "admin\_username" {

description = "Admin Username for VM"

type = string

default = "azureuser"

}

variable "admin\_password" {

description = "Admin Password for VM"

type = string

sensitive = true

}

✅ Variables allow easy configuration and reuse.

**Step 5: Define the Resource Group**

🔹 **File:** main.tf

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

name = var.resource\_group\_name

location = var.location

}

✅ This creates an **Azure Resource Group**.

**Step 6: Define the Virtual Network & Subnet**

🔹 **Add this to** main.tf

resource "azurerm\_virtual\_network" "vnet" {

name = "myVNet"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

address\_space = ["10.0.0.0/16"]

}

resource "azurerm\_subnet" "subnet" {

name = "mySubnet"

resource\_group\_name = azurerm\_resource\_group.rg.name

virtual\_network\_name = azurerm\_virtual\_network.vnet.name

address\_prefixes = ["10.0.1.0/24"]

}

✅ This creates a **VNet and Subnet**.

**Step 7: Define the Network Security Group (NSG)**

🔹 **Add this to** main.tf

resource "azurerm\_network\_security\_group" "nsg" {

name = "myNSG"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

security\_rule {

name = "AllowSSH"

priority = 1001

direction = "Inbound"

access = "Allow"

protocol = "Tcp"

source\_port\_range = "\*"

destination\_port\_range = "22"

source\_address\_prefix = "\*"

destination\_address\_prefix = "\*"

}

}

✅ This creates an **NSG allowing SSH (port 22) access**.

**Step 8: Define the Public IP & Network Interface**

🔹 **Add this to** main.tf

resource "azurerm\_public\_ip" "public\_ip" {

name = "myPublicIP"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

allocation\_method = "Dynamic"

}

resource "azurerm\_network\_interface" "nic" {

name = "myNIC"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

ip\_configuration {

name = "myNICConfig"

subnet\_id = azurerm\_subnet.subnet.id

public\_ip\_address\_id = azurerm\_public\_ip.public\_ip.id

private\_ip\_address\_allocation = "Dynamic"

}

}

✅ This configures a **Public IP and Network Interface (NIC)**.

**Step 9: Define the Azure Virtual Machine**

🔹 **Add this to** main.tf

resource "azurerm\_linux\_virtual\_machine" "vm" {

name = var.vm\_name

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

size = "Standard\_B1s"

admin\_username = var.admin\_username

admin\_password = var.admin\_password

disable\_password\_authentication = false

network\_interface\_ids = [

azurerm\_network\_interface.nic.id

]

os\_disk {

caching = "ReadWrite"

storage\_account\_type = "Standard\_LRS"

}

source\_image\_reference {

publisher = "Canonical"

offer = "UbuntuServer"

sku = "18.04-LTS"

version = "latest"

}

}

✅ This creates an **Ubuntu Virtual Machine**.

**Step 10: Define the Outputs**

🔹 **File:** outputs.tf

output "public\_ip" {

description = "Public IP of the VM"

value = azurerm\_public\_ip.public\_ip.ip\_address

}

✅ This will display the **Public IP of the VM** after deployment.

**Step 11: Initialize, Plan, and Deploy**

1️⃣ **Initialize Terraform**

terraform init

✅ Downloads required providers and initializes the working directory.

2️⃣ **Validate Configuration**

terraform validate

✅ Checks for syntax errors.

3️⃣ **Plan Execution**

terraform plan

✅ Shows what will be created.

4️⃣ **Apply Changes**

terraform apply -auto-approve

✅ Deploys the infrastructure in Azure.

**Step 12: Verify Deployment**

1️⃣ **Check deployed resources in Azure Portal** Go to [Azure Portal](https://portal.azure.com/) and navigate to **Resource Groups → myResourceGroup**.

2️⃣ **Get the VM's Public IP**

terraform output public\_ip

✅ Use this IP to **SSH into the VM**:

ssh azureuser@<public\_ip>

**Step 13: Destroy the Infrastructure**

If you want to delete all resources:

terraform destroy -auto-approve

✅ **Cleans up everything** from Azure.

✅ You have successfully **written and deployed an Azure Virtual Machine using Terraform**!

e.g. : main.tf fille look like below

provider "azurerm" {

features {}

}

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

name = "my-vm-rg"

location = "East US"

}

resource "azurerm\_virtual\_network" "vnet" {

name = "my-vnet"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

address\_space = ["10.0.0.0/16"]

}

resource "azurerm\_subnet" "subnet" {

name = "my-subnet"

resource\_group\_name = azurerm\_resource\_group.rg.name

virtual\_network\_name = azurerm\_virtual\_network.vnet.name

address\_prefixes = ["10.0.1.0/24"]

}

resource "azurerm\_network\_interface" "nic" {

name = "my-nic"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

ip\_configuration {

name = "internal"

subnet\_id = azurerm\_subnet.subnet.id

private\_ip\_address\_allocation = "Dynamic"

}

}

resource "azurerm\_virtual\_machine" "vm" {

name = "my-vm"

location = azurerm\_resource\_group.rg.location

resource\_group\_name = azurerm\_resource\_group.rg.name

network\_interface\_ids = [azurerm\_network\_interface.nic.id]

vm\_size = "Standard\_DS1\_v2"

storage\_os\_disk {

name = "osdisk"

caching = "ReadWrite"

create\_option = "FromImage"

managed\_disk\_type = "Standard\_LRS"

}

storage\_image\_reference {

publisher = "Canonical"

offer = "UbuntuServer"

sku = "18.04-LTS"

version = "latest"

}

os\_profile {

computer\_name = "myvm"

admin\_username = "adminuser"

admin\_password = "Password1234!"

}

os\_profile\_linux\_config {

disable\_password\_authentication = false

}

}

✅ You learned **Terraform basics to advanced** for **Azure**.

✅ Covered **Terraform structure, files, commands, & scripting**.

✅ Implemented a **real-world example** of deploying an **Azure VM**.

**Terraform Commands**

Terraform has several important commands used to manage infrastructure efficiently. Let's break them down into **categories**:

**1️⃣ Terraform Setup & Initialization Commands**

**1. terraform -v**

terraform -v

✅ **Checks the installed Terraform version.**

**2. terraform init**

terraform init

✅ **Initializes a Terraform working directory.**

🔹 Downloads required **provider plugins** (e.g., azurerm for Azure). 🔹 Initializes **Terraform backend** (if configured). 🔹 This command must be run **before applying any Terraform configuration**.

**3. terraform providers**

terraform providers

✅ **Lists all providers used in the configuration.**

🔹 Helps verify that Terraform is using the **correct provider versions**.

**4. terraform validate**

terraform validate

✅ **Checks for syntax errors in** .tf files.

🔹 Helps detect **typos, missing arguments, and invalid configurations** before deployment.

**2️⃣ Terraform Planning & Execution Commands**

**5. terraform plan**

terraform plan

✅ **Shows a preview of what Terraform will do.**

🔹 **Dry run** mode: Does not actually apply changes. 🔹 Displays which resources will be **created, modified, or destroyed**. 🔹 Helps avoid mistakes before running terraform apply.

**6. terraform apply**

terraform apply

✅ **Deploys the infrastructure as per** .tf configurations.

🔹 Prompts for confirmation before applying. 🔹 If you want to apply **without confirmation**, use:

terraform apply -auto-approve

🔹 This command **reads** .tf files and creates/modifies resources.

**7. terraform output**

terraform output

✅ **Displays output values defined in** outputs.tf.

🔹 Example output from outputs.tf:

output "resource\_group\_name" {

value = azurerm\_resource\_group.rg.name

}

terraform output resource\_group\_name

**3️⃣ Terraform State Management Commands**

**8. terraform state list**

terraform state list

✅ **Lists all managed resources in the Terraform state file.**

🔹 Useful to check what Terraform is managing.

**9. terraform state show lt;resourcegt;**

terraform state show azurerm\_resource\_group.rg

✅ **Displays detailed information about a specific resource.**

**10. terraform state rm lt;resourcegt;**

terraform state rm azurerm\_virtual\_machine.vm

✅ **Removes a resource from Terraform state (without deleting it in Azure).**

🔹 Terraform will **no longer track this resource**.

**11. terraform refresh (Deprecated – Use terraform apply instead)**

terraform refresh

✅ **Syncs the Terraform state file with actual resources in Azure.**

🔹 Ensures Terraform **knows the current state of resources**. 🔹 ⚠️ **Deprecated** in Terraform 1.6+ (Use terraform apply instead).

**4️⃣ Terraform Destroy & Cleanup Commands**

**12. terraform destroy**

terraform destroy

✅ **Deletes all Terraform-managed resources.**

🔹 If you want to destroy **without confirmation**, use:

terraform destroy -auto-approve

**\*\*13. terraform taint lt;resourcegt; (Deprecated – Use terraform apply -replace instead)**

terraform taint azurerm\_virtual\_machine.vm

✅ **Marks a resource for forced recreation.**

🔹 Deprecated in Terraform 1.0+. Instead, use:

terraform apply -replace=azurerm\_virtual\_machine.vm

**5️⃣ Terraform Import & Export Commands**

**14. terraform import lt;resourcegt; lt;Azure IDgt;**

terraform import azurerm\_resource\_group.rg /subscriptions/<sub\_id>/resourceGroups/my-rg

✅ **Imports an existing Azure resource into Terraform state.**

🔹 Only updates the state file, does not generate .tf files. 🔹 You need to manually write the .tf configuration afterward.

**15. terraform export (Experimental)**

terraform export

✅ **Exports existing infrastructure as Terraform code.** 🔹 **Still experimental**, may not work for all resources.

**6️⃣ Terraform Workspaces (Managing Multiple Environments)**

**16. terraform workspace list**

terraform workspace list

✅ **Lists all Terraform workspaces (environments).**

**17. terraform workspace new lt;namegt;**

terraform workspace new dev

✅ **Creates a new workspace (e.g.,** dev, prod).

🔹 Useful for managing different environments in a single Terraform setup.

**18. terraform workspace select lt;namegt;**

terraform workspace select dev

✅ **Switches to a different workspace.**

**7️⃣ Terraform Modules & Reusability**

**19. terraform fmt**

terraform fmt

✅ **Formats Terraform code to follow best practices.**

🔹 Ensures consistency in .tf files.

**20. terraform graph**

terraform graph

✅ **Generates a dependency graph of Terraform resources.**

🔹 Helps visualize the infrastructure.

**21. terraform show**

terraform show

✅ **Displays the Terraform state or execution plan in human-readable form.**

**8️⃣ Terraform Locking & Debugging**

**22. terraform lock**

terraform lock

✅ **Locks the Terraform state file to prevent simultaneous modifications.**

**23. terraform unlock**

terraform unlock

✅ **Removes the lock from the Terraform state file.**

**24. terraform console**

terraform console

✅ **Opens an interactive Terraform console for evaluating expressions.**

🔹 Example:

> length(["a", "b", "c"])

3

**25. terraform debug**

TF\_LOG=DEBUG terraform apply

✅ **Enables debug mode for troubleshooting.**

🔹 Use TF\_LOG=TRACE for **more detailed logs**.

**Terraform Providers & Modules – Structuring Your Code (Azure Cloud)**

🔹 **Imagine this:** You're deploying multiple **Azure resources**—Virtual Machines, Storage Accounts, Networking components, and Databases.

Now, **imagine doing this manually** every time for **different environments**—Development, Testing, and Production. Sounds **repetitive, time-consuming, and error-prone**, right?

**What if you could write once, reuse everywhere, and scale effortlessly?**

This is exactly what **Terraform Providers & Modules** help us achieve!

In this newsletter, we will cover:

✅ **Terraform Providers – Connecting Terraform to Azure**

✅ **Terraform Modules – Making Code Reusable & Scalable**

✅ **Step-by-Step Implementation with Azure**

**🔹 What is a Terraform Provider?**

A **Terraform Provider** is a **plugin** that allows Terraform to **interact with cloud services** (like Azure, AWS, GCP, Kubernetes, etc.).

For Azure, we use the **azurerm** provider to manage Azure resources.

🔹 **Example – Configuring Azure Provider in Terraform**

provider "azurerm" {

features {}

}

💡 **Real-World Use Case:**

A **DevOps team** managing Azure resources can use the **Azure provider** to:

✔ Automate the provisioning of Virtual Machines, Networking, and Storage

✔ Deploy Infrastructure across multiple environments **(Dev, QA, Prod)**

✔ Maintain consistency in cloud deployments

**Terraform Module:**

A Terraform module is a reusable, self-contained collection of Terraform configuration files that manage a specific piece of infrastructure.

It groups multiple resources together into a single logical unit that can be used and reused across different projects and environments.

Think of it as a **function in programming**—just like a function allows code reuse and modularity, a Terraform module enables infrastructure reuse and better organization.

**Why Use Terraform Modules in Azure?**

Using **Terraform modules** in **Azure** brings several benefits:

✅ **1. Reusability** Modules allow you to define Azure infrastructure once and reuse it across multiple projects/environments.

🔹 **Example**: Instead of writing the same **Azure Virtual Machine (VM) setup** multiple times, you create a reusable **Azure VM module**.

✅ **2. Maintainability** Organizing Terraform configurations into modules makes it easier to manage, update, and troubleshoot Azure infrastructure.

🔹 **Example**: If a **networking configuration** (Azure Virtual Network) needs to change, you only update the **VNet module** rather than modifying multiple configurations.

✅ **3. Scalability** Modules help scale infrastructure by allowing consistent provisioning of resources across different environments (e.g., Dev, QA, Prod).

🔹 **Example**: A **VNet module** can create different-sized networks dynamically based on input variables.

✅ **4. Consistency & Standardization** Using modules ensures Azure infrastructure follows best practices and compliance requirements.

🔹 **Example**: A **security module** can enforce specific **Azure Role-Based Access Control (RBAC)** policies across teams.

✅ **5. Easier Collaboration** Teams can work on different modules independently, improving collaboration and parallel development.

🔹 **Example**: The **networking team** manages the **Azure Network module**, while the **application team** configures the **Azure App Service module**.

✅ **6. Faster Deployment** Using **pre-built modules** speeds up Azure infrastructure provisioning, reducing deployment time.

**Structure of a Terraform Module**

A Terraform module typically consists of:

1. main.tf – Defines the primary resources (e.g., VMs, storage, networking).
2. variables.tf – Defines input variables to make the module configurable.
3. outputs.tf – Defines output values that expose key attributes of the resources.
4. providers.tf (Optional) – Defines cloud providers (e.g., AWS, Azure, GCP).

**🛠 Step-by-Step: Creating a Reusable Terraform Module for Azure Virtual Network, VM, and Storage**

Let’s **modularize** the deployment of an **Azure Virtual Network, VM, and Storage**.

**🔹 Step 1: Set Up Directory Structure**

Create the following folder structure:

terraform-modules/

├── modules/

│ ├── vnet/

│ │ ├── main.tf # Virtual Network resource

│ │ ├── variables.tf # Input variables

│ │ ├── outputs.tf # Outputs

│ ├── vm/

│ │ ├── main.tf # Virtual Machine resource

│ │ ├── variables.tf # Input variables

│ │ ├── outputs.tf # Outputs

│ ├── storage/

│ │ ├── main.tf # Storage Account resource

│ │ ├── variables.tf # Input variables

│ │ ├── outputs.tf # Outputs

├── main.tf # Calls the modules

├── variables.tf # Global variables

├── outputs.tf # Global outputs

├── providers.tf # Azure provider configuration

├── terraform.tfvars # Variable values

**🔹 Step 2: Configure the Provider**

Create providers.tf in the root directory:

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "~> 3.0"

}

}

}

provider "azurerm" {

features {}

}

👉 **Explanation:**

* This initializes **Terraform’s Azure provider**, which is required to manage Azure resources.
* The features {} block is necessary for some newer versions of AzureRM.

**🔹 Step 3: Create a Virtual Network Module**

Inside modules/vnet/, create main.tf, variables.tf, and outputs.tf.

**🔹 main.tf (VNet Module)**

resource "azurerm\_virtual\_network" "vnet" {

name = var.vnet\_name

location = var.location

resource\_group\_name = var.resource\_group\_name

address\_space = [var.address\_space]

}

👉 **Explanation:**

* Defines a **Virtual Network** (VNet) using azurerm\_virtual\_network.
* The **VNet name, location, resource group, and address space** are taken from variables.

**🔹 variables.tf (VNet Module)**

variable "vnet\_name" {}

variable "location" {}

variable "resource\_group\_name" {}

variable "address\_space" {}

👉 **Explanation:**

* Declares variables to make the module reusable.

**🔹 outputs.tf (VNet Module)**

output "vnet\_id" {

value = azurerm\_virtual\_network.vnet.id

}

👉 **Explanation:**

* Outputs the **VNet ID** so it can be referenced in other modules.

**🔹 Step 4: Create a Virtual Machine Module**

Inside modules/vm/, create main.tf, variables.tf, and outputs.tf.

**🔹 main.tf (VM Module)**

resource "azurerm\_virtual\_machine" "vm" {

name = var.vm\_name

location = var.location

resource\_group\_name = var.resource\_group\_name

network\_interface\_ids = [var.nic\_id]

vm\_size = var.vm\_size

storage\_os\_disk {

name = "${var.vm\_name}-os-disk"

caching = "ReadWrite"

create\_option = "FromImage"

managed\_disk\_type = "Standard\_LRS"

}

os\_profile {

computer\_name = var.vm\_name

admin\_username = var.admin\_username

admin\_password = var.admin\_password

}

os\_profile\_linux\_config {

disable\_password\_authentication = false

}

}

👉 **Explanation:**

* Defines an **Azure Virtual Machine (VM)**.
* The **network interface ID (nic\_id)** is passed as a variable to attach networking.
* Uses **variables for the VM name, size, username, and password**.

**🔹 variables.tf (VM Module)**

variable "vm\_name" {}

variable "location" {}

variable "resource\_group\_name" {}

variable "vm\_size" {}

variable "nic\_id" {}

variable "admin\_username" {}

variable "admin\_password" {}

**🔹 outputs.tf (VM Module)**

output "vm\_id" {

value = azurerm\_virtual\_machine.vm.id

}

👉 **Explanation:**

* Outputs the **VM ID** for reference in other resources.

**🔹 Step 5: Create a Storage Account Module**

Inside modules/storage/, create main.tf, variables.tf, and outputs.tf.

**🔹 main.tf (Storage Module)**

resource "azurerm\_storage\_account" "storage" {

name = var.storage\_account\_name

resource\_group\_name = var.resource\_group\_name

location = var.location

account\_tier = "Standard"

account\_replication\_type = "LRS"

}

**🔹 variables.tf (Storage Module)**

variable "storage\_account\_name" {}

variable "resource\_group\_name" {}

variable "location" {}

**🔹 outputs.tf (Storage Module)**

output "storage\_account\_id" {

value = azurerm\_storage\_account.storage.id

}

**🔹 Step 6: Call the Modules from Root**

Modify main.tf in the root directory to call these modules.

**🔹 main.tf (Root)**

module "vnet" {

source = "./modules/vnet"

vnet\_name = var.vnet\_name

location = var.location

resource\_group\_name = var.resource\_group\_name

address\_space = var.address\_space

}

module "vm" {

source = "./modules/vm"

vm\_name = var.vm\_name

location = var.location

resource\_group\_name = var.resource\_group\_name

vm\_size = var.vm\_size

nic\_id = "NIC\_ID" # Replace with actual value

admin\_username = var.admin\_username

admin\_password = var.admin\_password

}

module "storage" {

source = "./modules/storage"

storage\_account\_name = var.storage\_account\_name

resource\_group\_name = var.resource\_group\_name

location = var.location

}

👉 **Explanation:**

* Calls each module (vnet, vm, storage) and passes the required variables.

**🔹 Step 7: Define Variables in Root**

Modify variables.tf in the root directory.

variable "location" {}

variable "resource\_group\_name" {}

variable "vnet\_name" {}

variable "address\_space" {}

variable "vm\_name" {}

variable "vm\_size" {}

variable "admin\_username" {}

variable "admin\_password" {}

variable "storage\_account\_name" {}

**🔹 Step 8: Apply Terraform Configuration**

Run the following commands in the terminal:

terraform init # Initialize Terraform

terraform plan # Preview changes

terraform apply # Deploy infrastructure

* **Used Terraform modules** to create reusable Azure infrastructure components.
* **Configured Azure Virtual Network, VM, and Storage** as separate modules.
* **Implemented a modular, scalable, and maintainable structure** for Azure Terraform projects.

**🔹 Real-World Use Cases of Terraform Modules in Azure**

✔ **Multi-Environment Deployments:** Use modules to deploy identical infrastructure for **Dev, QA, and Prod**.

✔ **Infrastructure Standardization:** Teams can create reusable modules for **VMs, Networking, Storage**, ensuring consistency across projects.

✔ **Easier Maintenance:** Updating the module automatically applies changes across **all environments**.

**🔹 Key Takeaways**

✔ **Terraform Providers** allow Terraform to interact with cloud services like **Azure**.

✔ **Terraform Modules** help **organize and reuse code**, making infrastructure deployment **efficient**.

✔ **Modular Terraform Code** simplifies **multi-environment management**, reducing **errors** and increasing **automation**.

By implementing **Terraform Modules**, you can create **scalable, maintainable, and efficient** cloud infrastructure deployments.