***What is Terraform?***

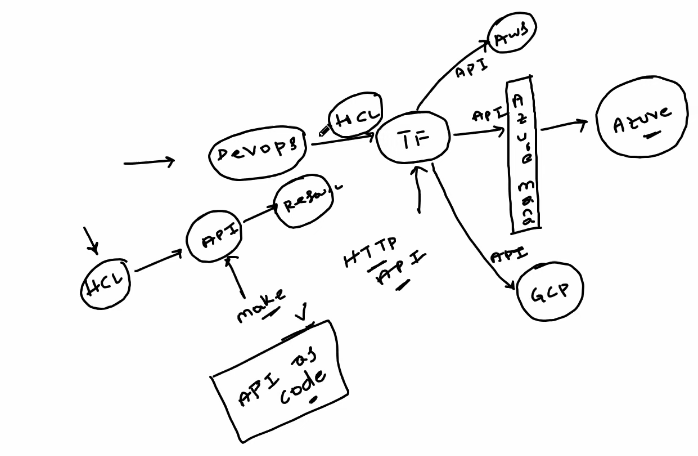
Terraform is an **open-source infrastructure as code (IaC) tool** developed by HashiCorp. It allows you to define, provision, and manage infrastructure resources across multiple cloud providers (like Azure, AWS, GCP) using a simple declarative configuration language (HCL - HashiCorp Configuration Language).

With Terraform, you write configurations that describe the desired state of your infrastructure. Terraform then translates these configurations into API calls to provision and manage the resources.

***Why Use Terraform Over Azure CLI or ARM Templates?***

While **Azure CLI** and **ARM Templates** are effective tools for managing Azure resources, Terraform offers several advantages:

**1. Multi-Cloud Support**

* **Azure CLI** and **ARM Templates** are Azure-specific tools.
* **Terraform** can manage resources across multiple cloud providers (Azure, AWS, GCP, etc.) and even on-premises infrastructure using the same workflow.
  + For example, you can manage Azure VMs and AWS S3 buckets within the same Terraform configuration.

**2. State Management**

* Terraform maintains a **state file** that tracks the current state of your infrastructure. This helps:
  + Detect changes in resources.
  + Plan and apply only the necessary updates.
* Azure CLI and ARM templates don't inherently have a similar built-in state management system, making incremental changes harder to track.

**3. Reusability and Modularity**

* Terraform allows you to create **modules** for reusable components. For example:
  + A module for creating a VNet.
  + A module for setting up a storage account.
* ARM Templates can be reused but are more complex to modularize, and Azure CLI scripts lack reusability mechanisms.

**4. Human-Readable Language**

* Terraform uses HCL, which is:
  + Simple, declarative, and easy to understand.
  + Easier to write and maintain than JSON-based ARM templates or scripting in Azure CLI.
* Example (Terraform HCL):

hcl

resource "azurerm\_virtual\_network" "example" {

name = "example-vnet"

location = "East US"

resource\_group\_name = "example-resources"

address\_space = ["10.0.0.0/16"]

}

**5. Execution Plans**

* Terraform shows a **detailed plan** before applying changes (terraform plan command). It tells you:
  + What resources will be added, updated, or destroyed.
* This ensures no surprises, which Azure CLI and ARM templates don't provide natively.

**6. Version Control and Collaboration**

* Terraform configurations can be stored in version control systems (like Git), enabling team collaboration.
* Azure CLI commands and ARM templates can also be stored in Git, but Terraform integrates better with tools like GitHub Actions, Jenkins, and Terraform Cloud for collaboration.

**7. Provider Ecosystem**

* Terraform supports many providers beyond Azure, including DNS providers, monitoring tools, and SaaS platforms.
* Azure CLI and ARM templates are strictly limited to Azure.

**Use Case Scenarios**

* **Azure CLI:** Best for quick, one-time tasks or scripts, such as deploying a single VM.
* **ARM Templates:** Ideal for Azure-specific, complex deployments where JSON is acceptable.
* **Terraform:** Best for managing infrastructure-as-code for multi-cloud environments, reusable components, and infrastructure lifecycle management.

***Commands:-***

**1.terraform init**

**Purpose:**

Initializes a Terraform working directory. It downloads provider plugins and sets up the backend for storing the Terraform state file. And help setup for backend state file

**Example:**

bash

terraform init

**Output:**

* Downloads the Azure provider (azurerm) if specified in the configuration.
* Prepares the working directory for other Terraform commands.

**2. terraform plan (dry run)**

**Purpose:**

Generates an execution plan, showing what Terraform will do without actually making any changes.

**Example:**

bash

terraform plan

**Output:**

* Shows the resources that will be created, updated, or deleted.
* Helps you review changes before applying them.

**3. terraform apply**

**Purpose:**

Applies the planned changes to create or modify infrastructure.

**Example:**

bash

terraform apply

**Interactive Confirmation:**

* Terraform asks for confirmation (type yes) before making changes.

***Variables:-***

**Using Variables in Terraform**

Terraform uses variables to make configurations dynamic. Here's how you can define and assign variables in Terraform:

**Step 1: Define a Variable**

Create a variables.tf file:

hcl

variable "location" {

description = "The location of the Azure resource"

default = "East US" # Optional, can be overridden

}

variable "resource\_group\_name" {

description = "The name of the resource group"

}

**Step 2: Reference Variables**

Use the variables in your Terraform configuration:

hcl

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

name = var.resource\_group\_name

location = var.location

}

**Step 3: Assign Variable Values**

You can assign variable values in multiple ways:

* **Via a terraform.tfvars file**:

hcl

location = "West Europe"

resource\_group\_name = "my-terraform-rg"

* **Via Command Line**:

bash

terraform apply -var="resource\_group\_name=my-terraform-rg" -var="location=West Europe"

* **Via Environment Variables**:

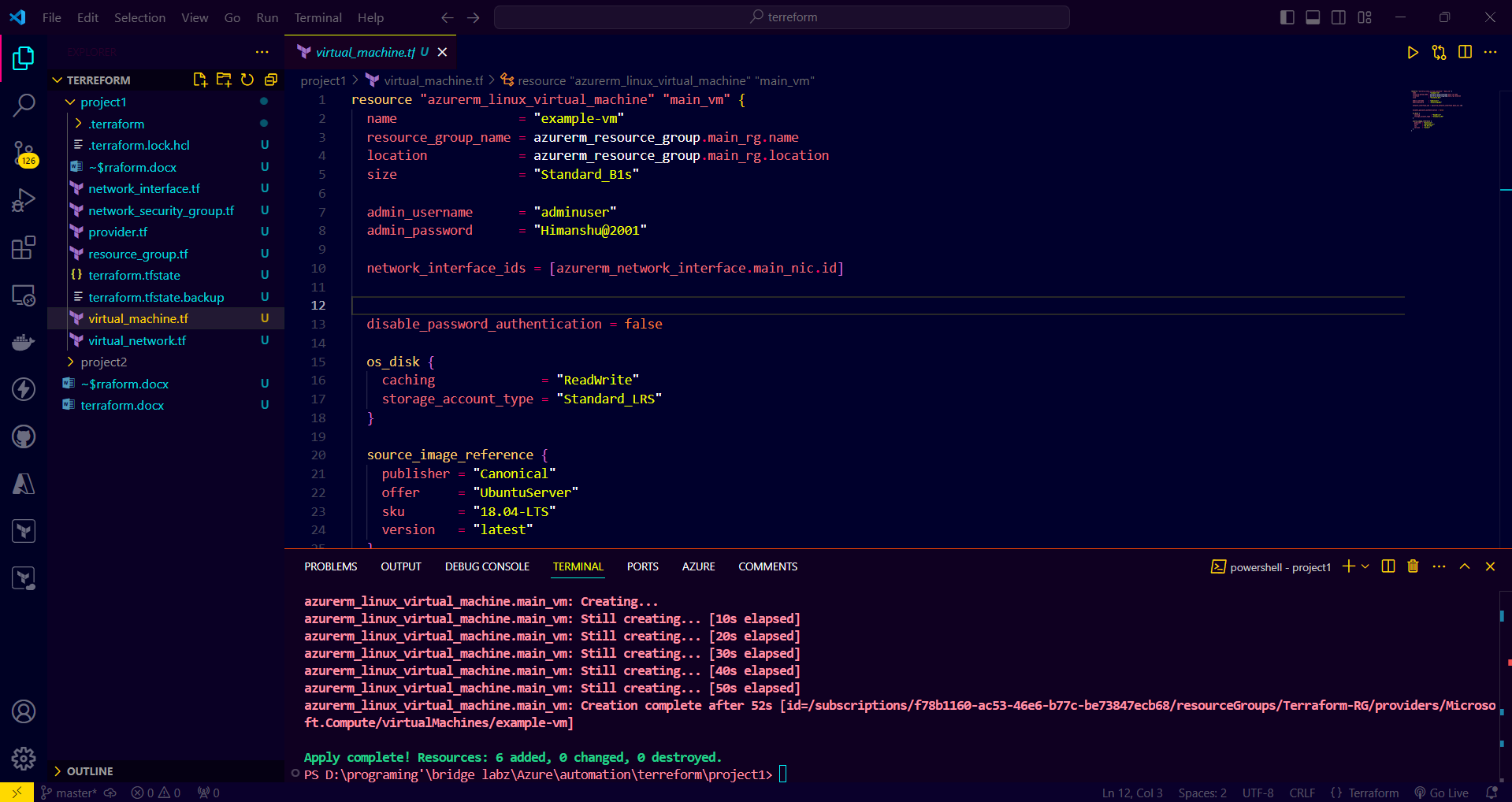
bash

export TF\_VAR\_location="West Europe"

export TF\_VAR\_resource\_group\_name="my-terraform-rg"

**Resource-Specific Requirement location and resource group**

Azure resources themselves require a location field in their API calls. While the resource group provides a location, Azure treats each resource as independent, and each resource needs to explicitly define a location to comply with Azure's API.



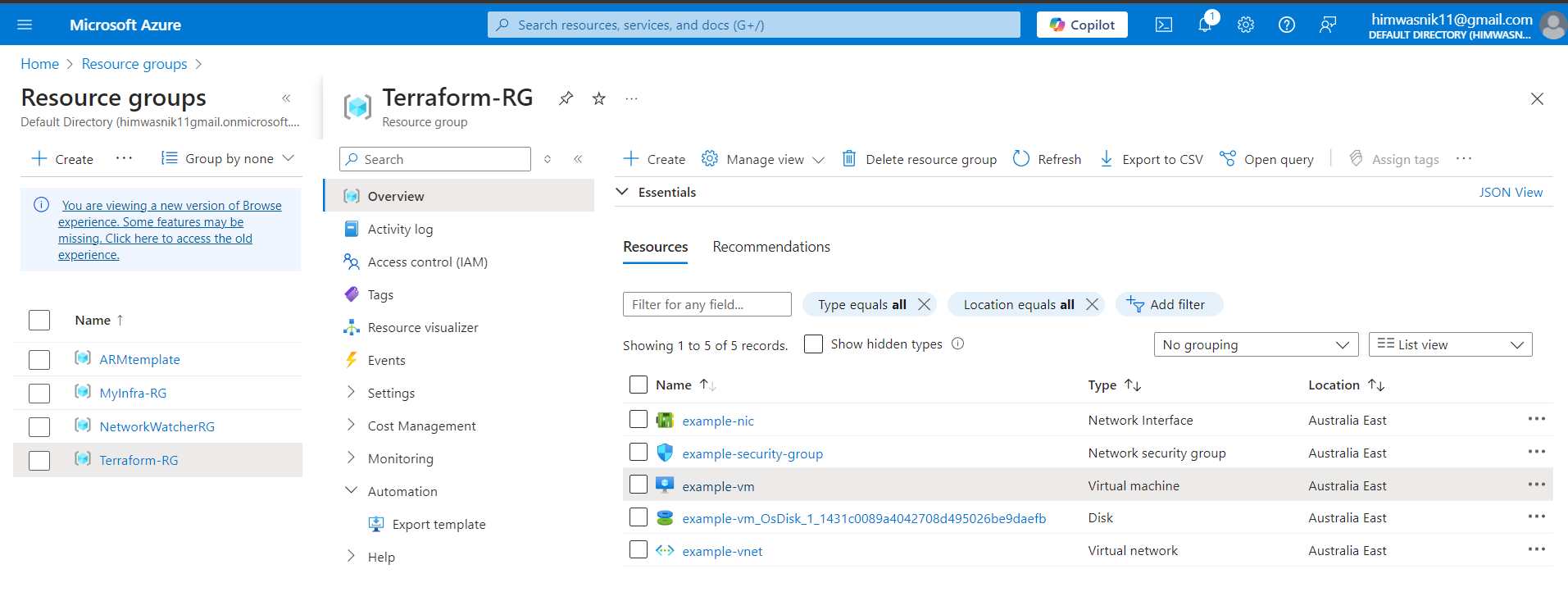


Fig 1.2 dynamically created

***Output:-***

The output.tf file is a convention used in Terraform to organize **output blocks**. These blocks are used to define the information you want to display after Terraform finishes creating or updating your infrastructure. Outputs are especially useful for exposing details like IP addresses, resource IDs, or URLs, which are needed for subsequent configurations or manual access.

output "public\_ip\_address" {

  value = data.azurerm\_public\_ip.existing\_public\_ip.ip\_address

}

output "azurerm\_network\_interface" {

  value = resource.azurerm\_network\_interface.example.name

}

***Terraform backend:-***

In Terraform, a **backend** is responsible for managing the **state file** of your infrastructure. It determines how Terraform stores, retrieves, and interacts with the state file during operations like terraform apply and terraform plan.

By default, Terraform uses a **local backend**, storing the state file (terraform.tfstate) on your local disk. However, you can configure remote backends to enable collaboration, security, and scalability.

* **Terraform backend configurations do not support using variables or resource references directly.** Backend configurations are processed before the main Terraform configuration, so you must provide static values in the backend block.

In **Terraform**, "state" refers to a file (terraform.tfstate) that tracks the resources managed by your configuration. This file is crucial for Terraform to map your infrastructure to your configuration.

There are two types of state management in Terraform:

**1. Local State**

* The state file is stored locally on your machine.
* It is the default behavior if no backend is configured.
* Typically stored as terraform.tfstate in the current working directory.

**Advantages:**

* **Simple Setup**: No additional configuration is needed.
* **Easy to Access**: The file is on your local system.

**Disadvantages:**

* **Single User**: Not suitable for collaboration; only one user can manage the infrastructure safely.
* **Risk of Loss**: If the file is deleted or corrupted, the state is lost.
* **No Versioning**: Changes to the state are not tracked.

**2. Remote State**

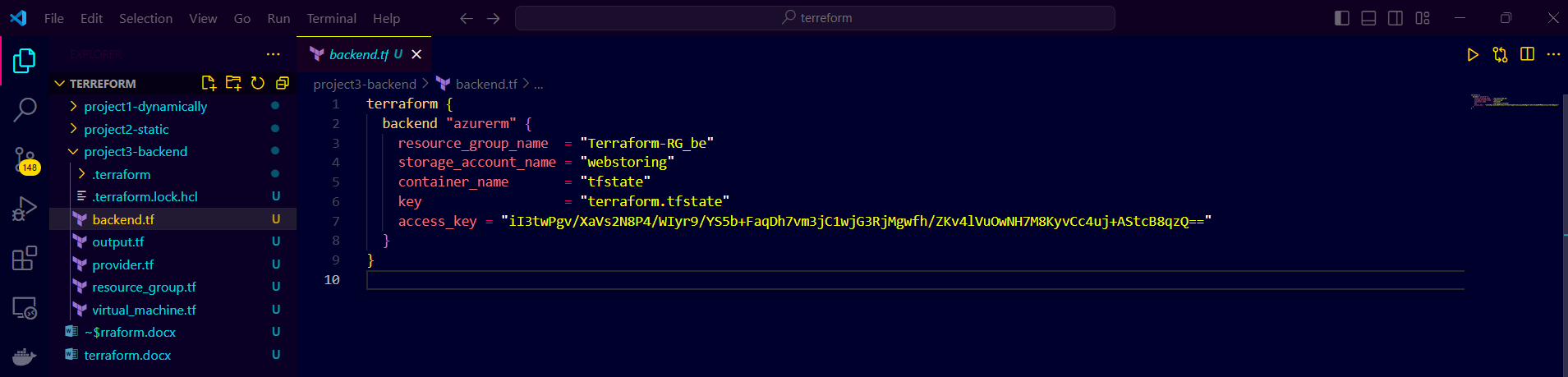
* The state file is stored in a remote backend such as Azure Storage, AWS S3, Google Cloud Storage, or Terraform Cloud.
* Requires backend configuration in your Terraform configuration.

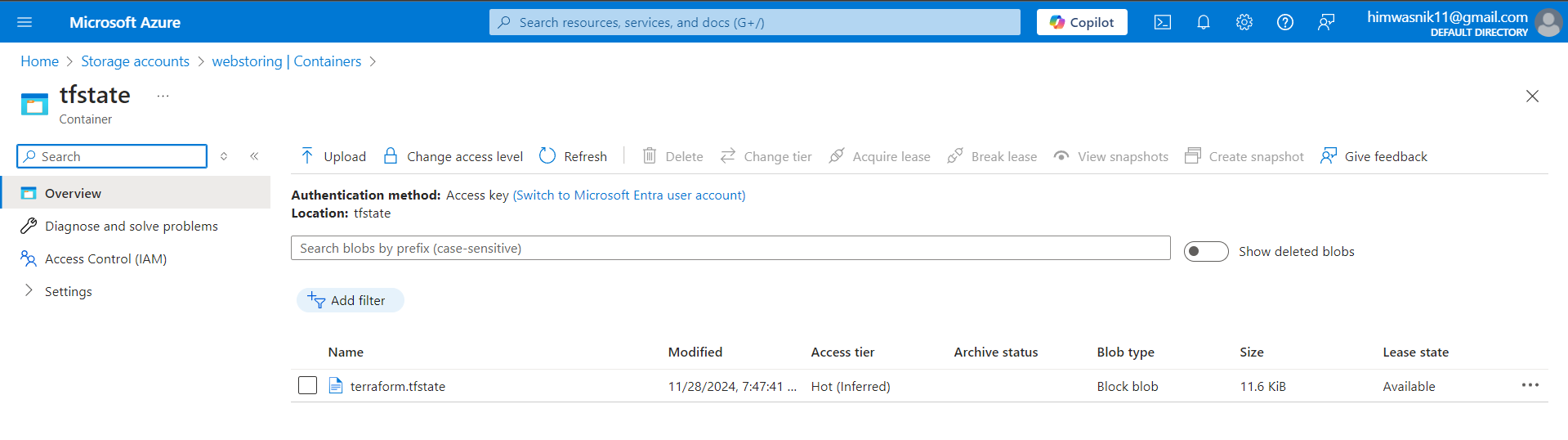
**Advantages:**

* **Collaboration**: Multiple users can access the state simultaneously.
* **Versioning**: Many backends (e.g., Azure, S3) support state versioning, making it easier to recover from errors.
* **Centralized Management**: State is stored in a secure, central location.
* **Locking**: Remote backends often provide locking to prevent multiple users from making conflicting updates.

**Disadvantages:**

* **Setup Complexity**: Requires configuring and securing a backend.
* **Dependency**: Requires access to the remote backend, which may not be ideal in offline scenarios.

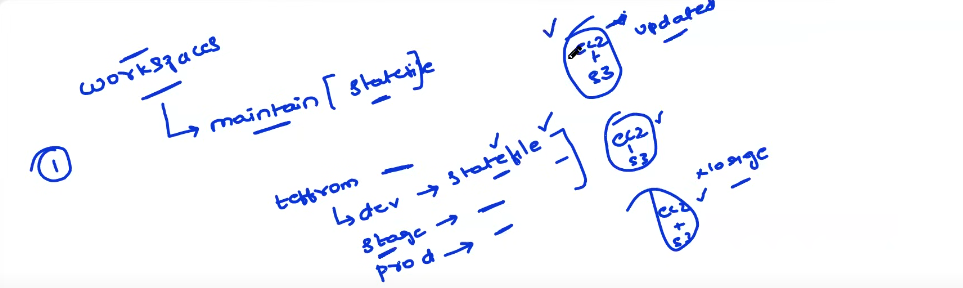




***Terraform Workspaces in Azure***

**Terraform Workspaces** allow you to manage multiple environments (e.g., dev, test, prod) using the same Terraform configuration. Each workspace has its own **state file**, enabling you to create and manage resources in separate environments without needing separate configuration files.

Here’s how you can use **workspaces** in an Azure environment.



**Azure Virtual Network Deployment in Two Environments**

* **Development Workspace**: This workspace will deploy a virtual network in the Azure Dev environment.
* **Production Workspace**: This workspace will deploy a virtual network in the Azure Production environment.

**Steps to Implement the Scenario:**

**Step 1: Install Terraform**

If you haven’t already installed Terraform, start by doing so.

1. **Install Terraform**: Download it from Terraform's official website.
2. **Verify installation**: Run the following command to check if Terraform is installed correctly:

bash

terraform --version

**Step 2: Initialize Your Project Directory**

Create a project directory for the Terraform configuration.

1. Create a directory for your project:

bash

mkdir terraform-azure-workspaces

cd terraform-azure-workspaces

1. Inside this directory, create a file called main.tf where you’ll define your resources (e.g., Virtual Network).

**Step 3: Set Up Terraform Provider**

In the main.tf file, we will set up the **Azure Provider**. This tells Terraform how to interact with Azure.

**Step 4: Define the Virtual Network Resource**

Below the provider, define a **Virtual Network** resource that will be used by both the development and production workspaces. This is a basic virtual network with configurable name, address space, and location.

Hcl

terraform {

  required\_providers {

    azurerm = {

      source  = "hashicorp/azurerm"

      version = "~> 3.0"

    }

  }

}

provider "azurerm" {

  features {}

}

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

  name     = "rg-${terraform.workspace}"

  location = "East US"

}

resource "azurerm\_virtual\_network" "example" {

  name                = "vnet-${terraform.workspace}"

  location            = azurerm\_resource\_group.example.location

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

  address\_space       = ["12.0.0.0/20"]

}

The ${terraform.workspace} variable dynamically names the virtual network based on the current workspace.

**Step 5: Initialize Terraform Configuration**

Before you begin using workspaces, initialize your Terraform configuration to download the necessary provider plugins.

bash

terraform init

**Step 6: Create and Switch Workspaces**

Now, we’ll create two workspaces: one for **development** and one for **production**.

1. **Create and switch to the development workspace**:

bash

terraform workspace new development

1. **Create and switch to the production workspace**:

bash

terraform workspace new production

You can view the active workspace using:

bash

terraform workspace show

**Step 7: Plan and Apply the Infrastructure for Each Workspace**

**For Development Workspace:**

1. Switch to the **development** workspace if you are not already in it:

bash

terraform workspace select development

1. **Plan** the infrastructure:

bash

terraform plan

1. **Apply** the configuration to deploy the resources:

bash

terraform apply

Terraform will deploy the virtual network with a name like example-vnet-development.

**For Production Workspace:**

1. Switch to the **production** workspace:

bash

terraform workspace select production

1. **Plan** the infrastructure:

bash

terraform plan

1. **Apply** the configuration to deploy the resources:

bash

terraform apply

Terraform will deploy a virtual network with a name like example-vnet-production.

**Step 8: Verify the Resources**

After applying both workspaces, you can go to the **Azure portal** and verify that two virtual networks have been created under different names:

* example-vnet-development
* example-vnet-production

**Step 9: Clean Up Resources**

If you want to destroy the resources in either workspace, follow these steps:

1. Select the workspace you want to clean up:

bash

terraform workspace select development

1. Destroy the resources:

bash

terraform destroy

Repeat for the **production** workspace.

**Summary of the Steps**

1. **Install Terraform** and initialize the project.
2. **Create the main.tf file** with the necessary provider configuration and resources.
3. **Set up two workspaces** (development and production).
4. **Deploy resources** in each workspace using terraform apply.
5. **Clean up** by destroying the resources when done.

This is a simple scenario where workspaces allow you to manage different environments (Dev and Prod) without duplicating your Terraform configuration. You can also configure different variables for each workspace if needed.

Here’s a comprehensive list of **Terraform workspace commands** and their purposes, along with examples for common workspace-related tasks. Workspaces in Terraform allow you to manage multiple environments (e.g., dev, staging, production) within the same configuration.

***Workspace Commands***

**1. View the Current Workspace**

bash

terraform workspace show

* Displays the name of the currently active workspace.

**2. List All Workspaces**

bash

terraform workspace list

* Lists all existing workspaces.
* The currently active workspace will be marked with an asterisk (\*).

**3. Create a New Workspace**

terraform workspace new <workspace\_name>

* Creates a new workspace with the specified name.
* Example:

bash

terraform workspace new staging

**4. Switch to an Existing Workspace**

terraform workspace select <workspace\_name>

* Switches to the specified workspace.
* Example:

bash

terraform workspace select production

**5. Delete an Existing Workspace**

bash

terraform workspace delete <workspace\_name>

* Deletes the specified workspace.
* Note: You cannot delete the currently active workspace.
* Example:

terraform workspace delete dev

**6. Create and Switch to a Workspace (Combined)**

bash

terraform workspace new <workspace\_name>

* If the workspace doesn't exist, it creates and switches to it in one step.