Hewlett Packard Enterprise

Terraform Training - Module 1

Hewlett Packard Enterprise

Chapter 1: Terraform Introduction

1.1 Course Concepts

This module covers many topics that are necessary for using Terraform proficiently.

- What is Terraform?
  - Infrastructure as Code Concepts and Use Cases
  - Terraform vs Other DevOps Tools
- Basic Terraform Use
  - Terraform Core Components
  - Creating and Accessing Cloud Resources on Azure
  - Terraform Syntax, Internals and Patterns
  - Modifying Infrastructure
  - Dependencies
- Practical Terraform Use
  - Working with Remote State Backends
  - Authoring and Using Terraform Modules

1.2 Terraform Overview

There are a few different flavors of Terraform available from HashiCorp:

- **Community**:
  The community version is shipped as a single binary written in Go. It's a command line application, but very easy to get started using.
  - Always free.
  - Users need to self-managed.
  - The code is separated as Terraform core, which provides only the necessary functionality to support the Terraform workflows. Interfaces to various service providers such as Amazon Web Services, Google Cloud Platform, and Microsoft Azure are separate as part of a large ecosystem of plugins and providers.
- **Terraform Cloud**:
  - There are three Edition:
    - Free Edition: Get started with all capabilities needed for infrastructure-as-code provisioning. The first 500 resources per month are free.
    - Standard Edition: For professional individuals or teams adopting infrastructure as code provisioning. Pay as you go, and Enterprise support included.
    - Plus Edition: For enterprises standardizing and managing infrastructure automation and lifecycle, with scalable runs. Contact Terraform sales for custom price and Enterprise support included.
  - SaaS version, managed by Terraform.
- **Enterprise**:
  - For enterprises with special security, compliance, and additional operational requirements.
  - Contact Terraform sales for custom price and Enterprise support included.
  - Users need to be self-managed.

1.3 What is Infrastructure as Code?

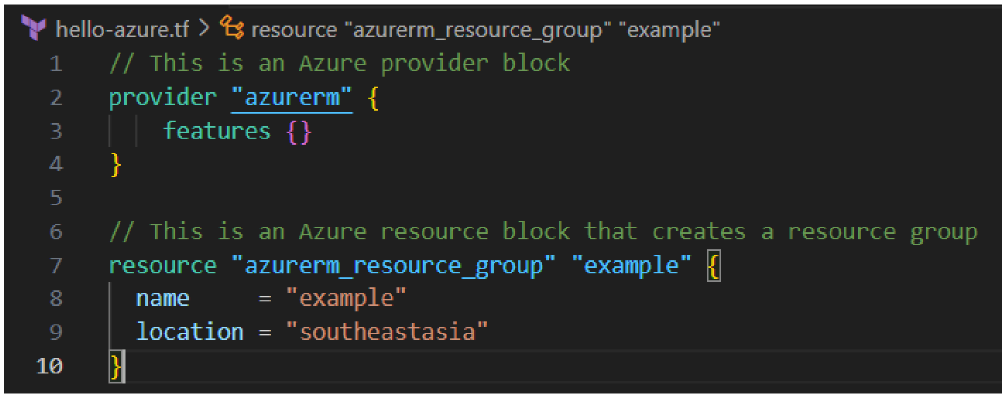
- Infrastructure as Code is a collection of methodologies for writing configuration or scripts which deploy and manage infrastructure. One of the key enablers of Infrastructure as Code is the prevalence of virtualization across many different areas of the hardware stack, including virtual machines, software defined networking, and storage offerings.
- There are many different tools available for Infrastructure as Code (IaC)

  - Terraform
  - Deployment Manager (Azure)
  - Ansible
  - Puppet
  - Chef
  - SaltStack
- Each of these tools has nuances which solve different challenges in the IaC ecosystem.

1.4 Example

As an example, the following block of Terraform code creates a resource group on Azure. Since code is a simple text file, it can be versioned and tracked easily, which is a big contrast to manually provisioning and managing servers.

We will use hello-azure.tf throughout this module to explore the different fundamental concepts in Terraform.


hello-azure.tf > resource "azurerm_resource_group" "example"

```
// This is an Azure provider block
provider "azurerm" {
    features {}
}

// This is an Azure resource block that creates a resource group
resource "azurerm_resource_group" "example" {
  name     = "example"
  location = "southeastasia"
}
```

1.5 Infrastructure as Code Pipeline

The ability to provision infrastructure using code rather than slow and manual processes facilitates the adoption of battle-tested software development practices for building out infrastructure as code pipelines.

Furthermore, infrastructure as code makes it easier to set up identical copies of infrastructure using Infrastructure as Code. Being able to do this opens the door to new testing strategies, including ephemeral test environments for software applications, and new architectures for resiliency (such as deploying identical copies of infrastructure in two different data centers without any code changes).

1.6 Why Infrastructure as Code

Modern virtualization and cloud providers have drastically reduced the procurement time for various types of resources. This presents several challenges as well as advantages. Take some time to consider how these different features of infrastructure as code have already or would affect your existing technology workflows.

- Versioning
  - The code that defines infrastructure can be versioned, so it’s easy to review who made changes, what they changed, and when those changes were made.
- Modularity
  - The same code can be used to provision the exact same set of virtual infrastructure across geographic regions and in different data centers. It’s also possible to modularize common design patterns so that a developer, for example, can have the relevant part of the infrastructure configured in her sandbox environment.

- There are also many open-source modules available that provide a higher level of abstraction of different components of various infrastructure deployments.
- Composability
    - Different modules can be easily pieced together in different ways and shared with others simply by sharing the code.
    - Well-parameterized code can easily plug into different types of configurations, or even different use cases altogether.
- Rollback or Restore
    - With the infrastructure defined as code, it makes it easier to roll back to a previously tested state.
- Full stack dev/deployment
    - Developers can define their infrastructure requirements as code and own the entire stack. It facilitates more complete testing by deploying ephemeral infrastructure environments, as well as a documented, auditable configuration as code is promoted through environments to production.
- Enables self-service infrastructure.
    - Help desk tickets to provision infrastructure are greatly reduced. With a common infrastructure as code tool across an organization, it becomes easier to provide the infrastructure that different groups need by allowing them to provision certain types of resources rather than manually reviewing, parsing and acting upon a help desk ticket.
- Continuous Provisioning
    - As Infrastructure as code pipelines become more sophisticated, fully automated testing and deployment can speed the development cycle. Infrastructure that is automatically provisioned by a trigger (for example, a merge in a version control system like git) is known as continuous provisioning.
- Immutable infrastructure
    - Since it’s easy to reproduce infrastructure in a known and tested state, it becomes less and less useful to upgrade virtual infrastructure in place. There’s always some configuration drift that could make an in-place upgrade go awry, and adopting a philosophy of immutable infrastructure is one strategy for preventing those issues.
- New automation and deployment workflows for infrastructure

1.7 Infrastructure as Code Use Cases

- **Multi-Tier Applications**
    - A collection of resources can be built and managed with a single tool while tracking the dependencies between each tier (web servers, application servers, load balancers, databases, etc.)
- **Multi-cloud, Multi-region, or Multi-environment Deployment**
    - Infrastructure as code tools that support multiple cloud or other service providers make it easier to leverage the best features for each cloud.
    - Dev environments can be structured and built with code then shared with other teams and/or leveraged for other environments. Think Quality Assurance, other Development teams and replicating to staging environments in separate regions and/or increasing fault tolerance with redundant regions/cloud.

1.8 Features of Terraform

- **Extensible**
    - Extensions which provide entry point for creating resources are called providers.
- **Modular**
    - Plugin-based architecture provides other powerful methods of customization.

- **Composable**: Allows new infrastructure layouts to reference existing infrastructure.
  - Dependent infrastructure can be loosely coupled.
  - Create higher level abstractions of infrastructure patterns.
- **Portable:**
  - Terraform is a single binary and downloads dependencies via **terraform init**
- **Declarative:**
  - Rather than writing instructions, describe what you want and terraform will figure out how to get there.
- **Provisioning:**
  - Updating, decommissioning infrastructure and related components
- **Agentless:**
  - No installation of software required on target hosts.

1.9 Why Use Terraform vs Something Else?

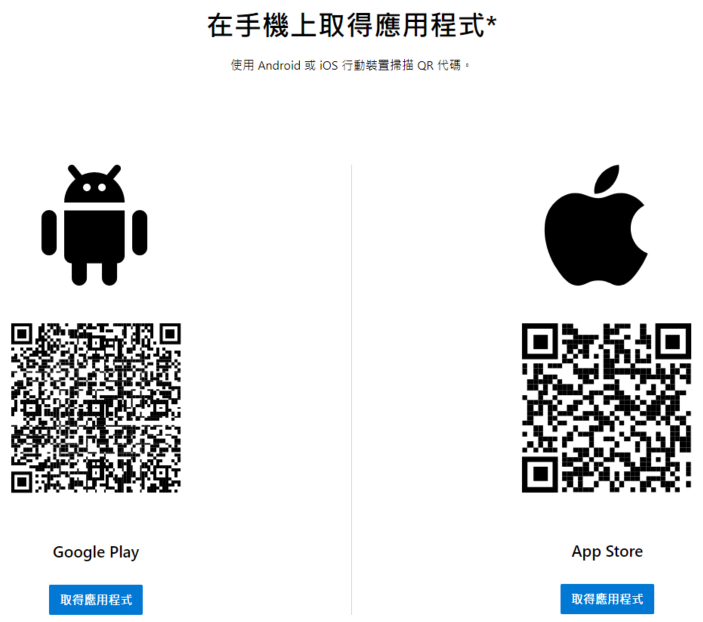
Terraform is one tool in a rich ecosystem of DevOps tools. One thing that Terraform does well is create and provision infrastructure. However, depending on the type of provider, creating, or altering or destroying can be time consuming, so it may not always be the right tool for every job.

- Configuration Management
  - Some other tools are aimed specifically at configuration management, which are useful for “last mile” configuration after using Terraform, or when an upgrade or alteration of an existing piece of infrastructure needs to occur.
    - Puppet
    - Chef
    - Ansible
- Agentless vs Agent-based
  - Terraform requires no software agent to be installed on the target resources. This means resources can be a multitude of different types of abstractions, from virtual machines to managing a database.
  - If terraform can’t reach the infrastructure resources (via SSH or some other secure path), it might require destroying and rebuilding that infrastructure, and in some cases is not feasible. An agent-based configuration management tool provides an alternative way for resources to access the configuration rather than having to expose a channel for an external tool to connect to the resource.
- Declarative vs Imperative: Declarative means describing the desired end state whereas imperative means writing instructions to get to the state. Terraform is declarative in nature, and the code often maps well to an existing architecture or design diagram for infrastructure.
- Terraform is Cloud/Infrastructure agnostic which enables hybrid IT.
  - Agnosticism is built through full capability providers and allows extension through implementation of custom providers.
- Terraform has an active open-source community that also includes enterprises who maintain official providers.
- Immutable infrastructure vs Mutable Infrastructure

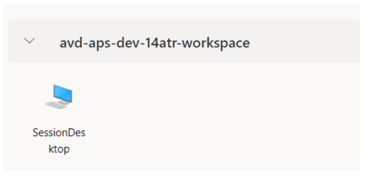
Lab 1: Setup your environment

In this lab, we will set up the terraform environment on AVD (Azure Virtual Desktop).

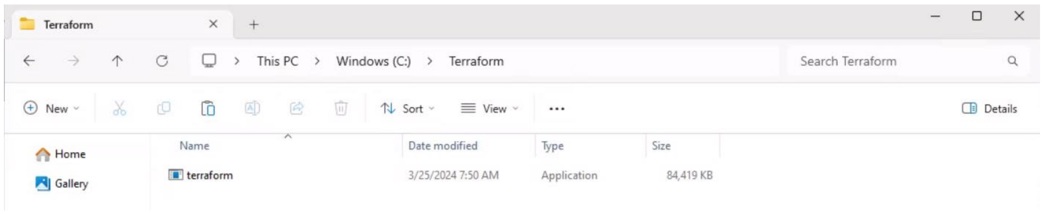
1. Install the Authenticator on your mobile device.



2. Using the link to connect to AVD through the browser. (If you don't want to switch the user, you can use the incognito window) https://client.wvd.microsoft.com/arm/webclient/index.html
3. Login with your username and temporary password, then change your password.

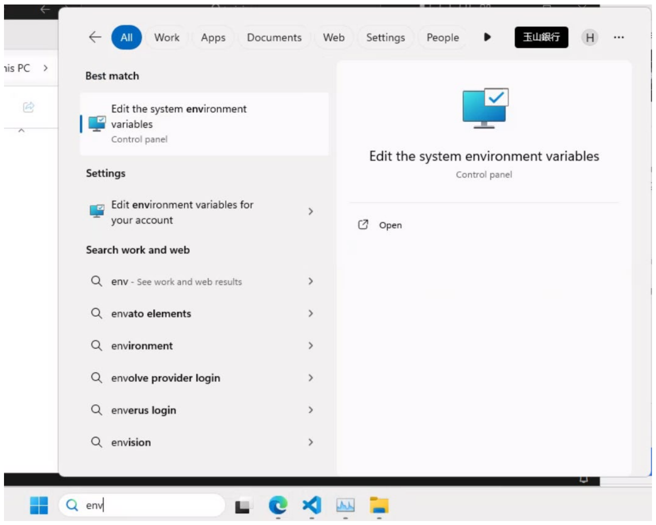


4. Click the "SessionDesktop" then you can find the windows desktop.
5. Install the Visual Studio Code. https://code.visualstudio.com/
6. Download the Terraform. https://developer.hashicorp.com/terraform/install?ajs_aid=02461d35-2348-404e-8568-fce1974f0565&product_intent=terraform#windows
7. Put the terraform.exe file to the folder, for example: "C:\Terraform".

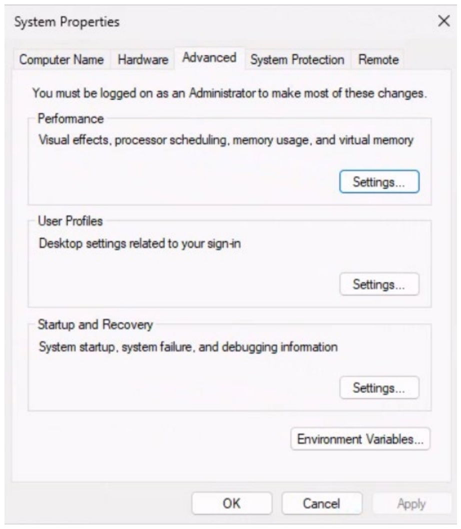


Name	Date modified	Type	Size
terraform	3/25/2024 7:50 AM	Application	84,419 KB

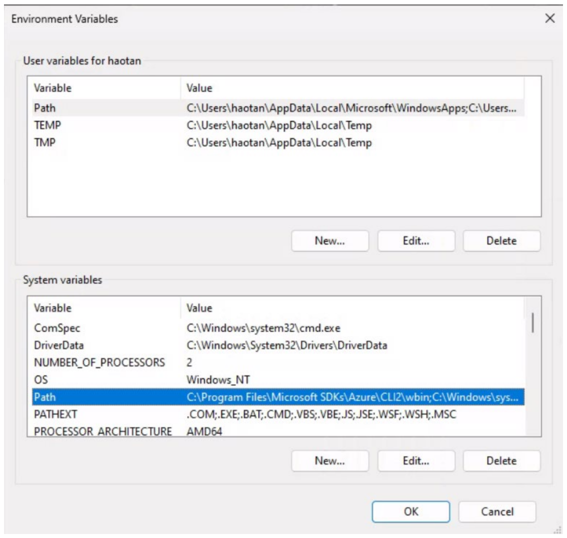
8. Open the “Edit the system environment variables”.



9. Click “Environment Variable”.


System Properties
Computer Name
Hardware
Advanced
System Protection
Remote
You must be logged on as an Administrator to make most of these changes.
Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...
User Profiles
Desktop settings related to your sign-in
Settings...
Startup and Recovery
System startup, system failure, and debugging information
Settings...
Environment Variables...
OK
Cancel
Apply

10. Select the “Path” variable in System variables, then click “Edit”.


Environment Variables

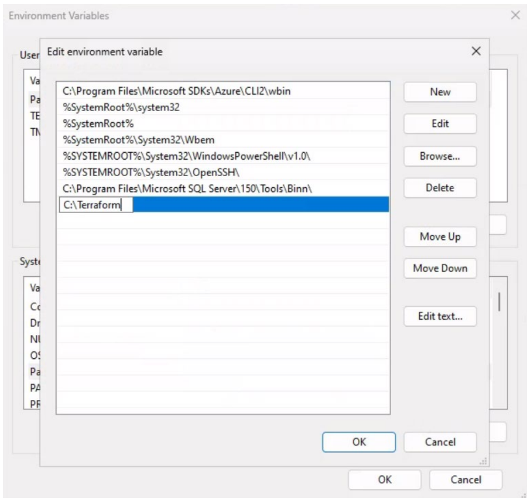
User variables for haotan

Variable	Value
Path	C:\Users\haotan\AppData\Local\Microsoft\WindowsApps;C:\Users...
TEMP	C:\Users\haotan\AppData\Local\Temp
TMP	C:\Users\haotan\AppData\Local\Temp

System variables

Variable	Value
ComSpec	C:\Windows\system32\cmd.exe
DriverData	C:\Windows\System32\Drivers\DriverData
NUMBER_OF_PROCESSORS	2
OS	Windows_NT
Path	C:\Program Files\Microsoft SDKs\Azure\CLI2\wbin;C:\Windows\sys...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC
PROCESSOR_ARCHITECTURE	AMD64

11. Click “New” and enter the path which terraform.exe located. Then click “OK”.

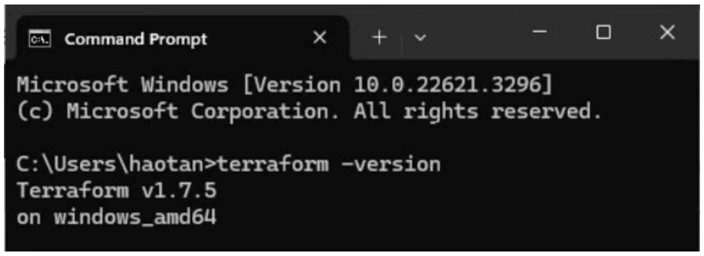


Environment Variables

Edit environment variable

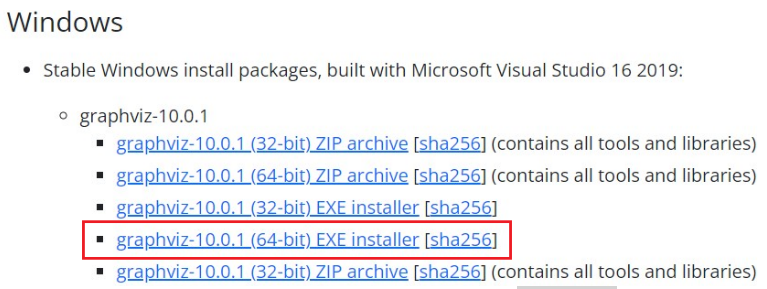
```
C:\Program Files\Microsoft SDKs\Azure\CLI2\wbin
%SystemRoot%\system32
%SystemRoot%
%SystemRoot%\System32\Wbem
%SYSTEMROOT%\System32\WindowsPowerShell\v1.0\
%SYSTEMROOT%\System32\OpenSSH\
C:\Program Files\Microsoft SQL Server\150\Tools\Binn\
C:\Terraform
```

12. Open cmd, then execute the terraform -version to check if its success or not.



```
Microsoft Windows [Version 10.0.22621.3296]
(c) Microsoft Corporation. All rights reserved.

C:\Users\haotan>terraform -version
Terraform v1.7.5
on windows_amd64
```

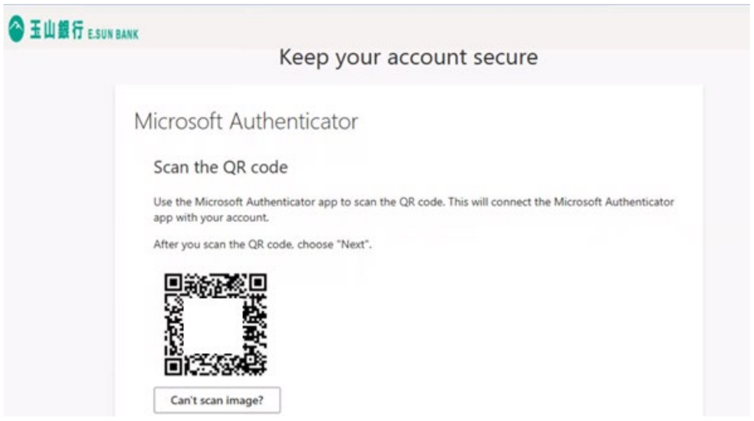
13. Install Azure CLI. https://learn.microsoft.com/zh-tw/cli/azure/install-azure-cli-windows?tabs=azure-cli
14. Install Graphviz. https://graphviz.org/download/



Windows

- Stable Windows install packages, built with Microsoft Visual Studio 16 2019:
  - graphviz-10.0.1
    - graphviz-10.0.1 (32-bit) ZIP archive [sha256] (contains all tools and libraries)
    - graphviz-10.0.1 (64-bit) ZIP archive [sha256] (contains all tools and libraries)
    - graphviz-10.0.1 (32-bit) EXE installer [sha256]
    - graphviz-10.0.1 (64-bit) EXE installer [sha256]
    - graphviz-10.0.1 (32-bit) ZIP archive [sha256] (contains all tools and libraries)

15. Open the CMD, the execute the command **az login**. Then it will open a browser and ask you to login.
16. Scan the QR code on the browser, then add to your Authenticator.



Keep your account secure

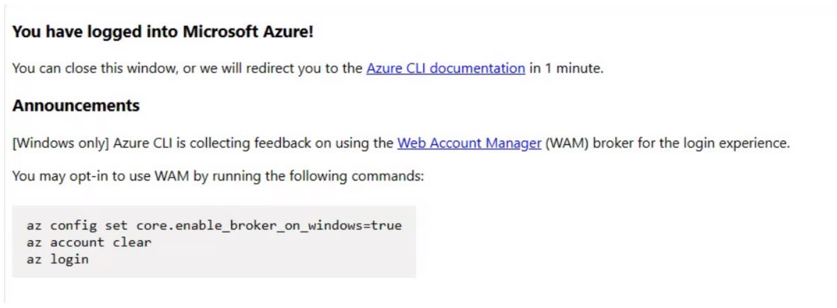
Microsoft Authenticator

Scan the QR code

Use the Microsoft Authenticator app to scan the QR code. This will connect the Microsoft Authenticator app with your account.

After you scan the QR code, choose "Next".

17. If login succeeded, you would see below screenshot.



You have logged into Microsoft Azure!

You can close this window, or we will redirect you to the Azure CLI documentation in 1 minute.

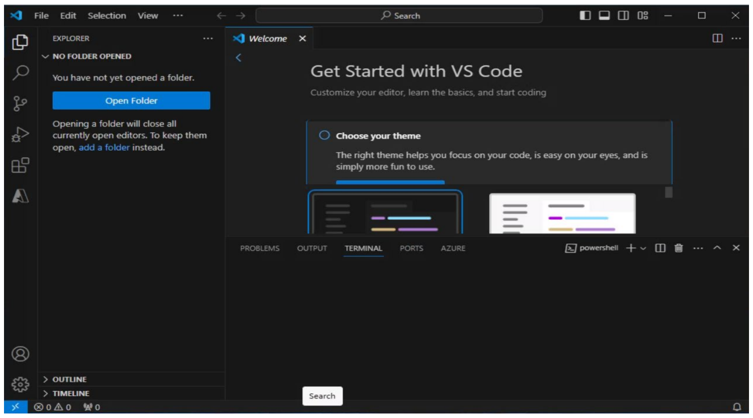
Announcements

[Windows only] Azure CLI is collecting feedback on using the Web Account Manager (WAM) broker for the login experience.

You may opt-in to use WAM by running the following commands:

```
az config set core.enable_broker_on_windows=true
az account clear
az login
```

18. Open the Visual Studio Code.



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

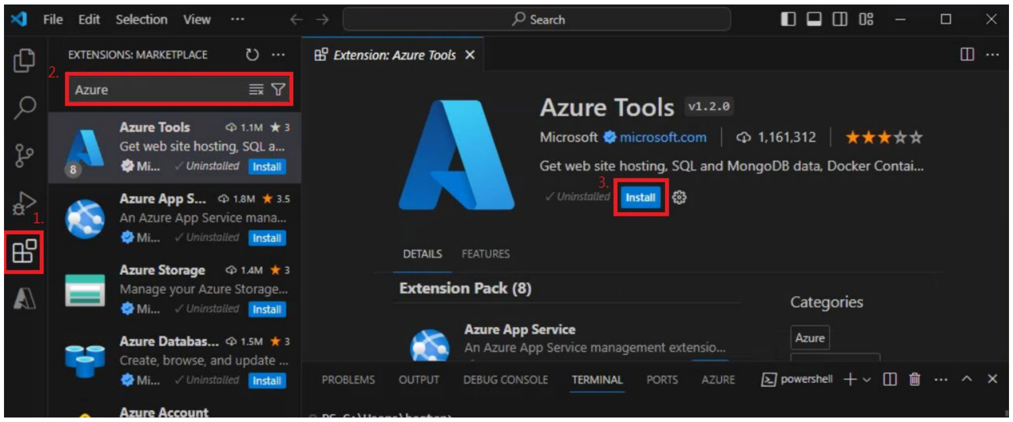
Get Started with VS Code

Customize your editor, learn the basics, and start coding

Choose your theme

The right theme helps you focus on your code, is easy on your eyes, and is simply more fun to use.

19. Install the extension to make your work easier. For example: Azure Tools, Azure CLI Tools and HashiCorp Terraform.



Azure Tools 1.1M ★ 3 Get web site hosting, SQL a... Mi... ✓ Uninstalled Install

Azure App S... 1.8M ★ 3.5 An Azure App Service mana... Mi... ✓ Uninstalled Install

Azure Storage 1.4M ★ 3 Manage your Azure Storage... Mi... ✓ Uninstalled Install

Azure Databas... 1.5M ★ 3 Create, browse, and update ... Mi... ✓ Uninstalled Install

Azure Account

Azure Tools v1.2.0

Microsoft microsoft.com | 1,161,312 | ★★★☆☆

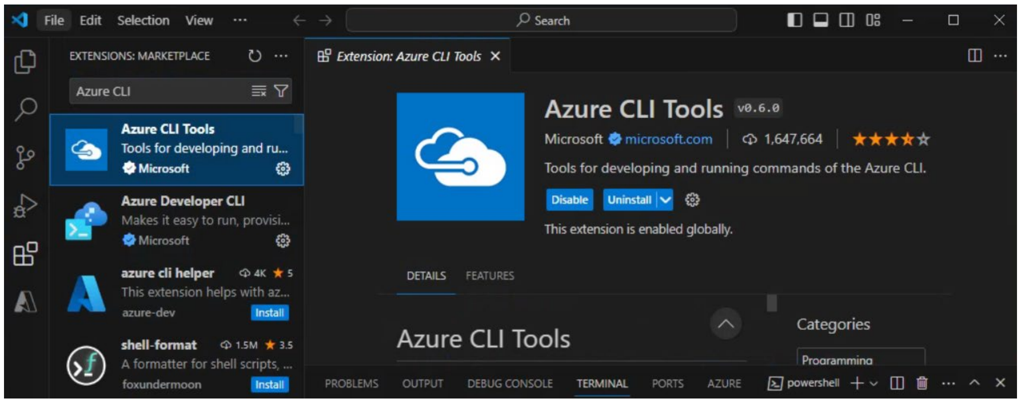
Get web site hosting, SQL and MongoDB data, Docker Contai...

✓ Uninstalled Install

Extension Pack (8)

Azure App Service

An Azure App Service management extensio...



Azure CLI Tools Tools for developing and ru... Microsoft

Azure Developer CLI Makes it easy to run, provisi... Microsoft

azure cli helper 4K ★ 5 This extension helps with az... azure-dev Install

shell-format 1.5M ★ 3.5 A formatter for shell scripts, ... foxundermoon Install

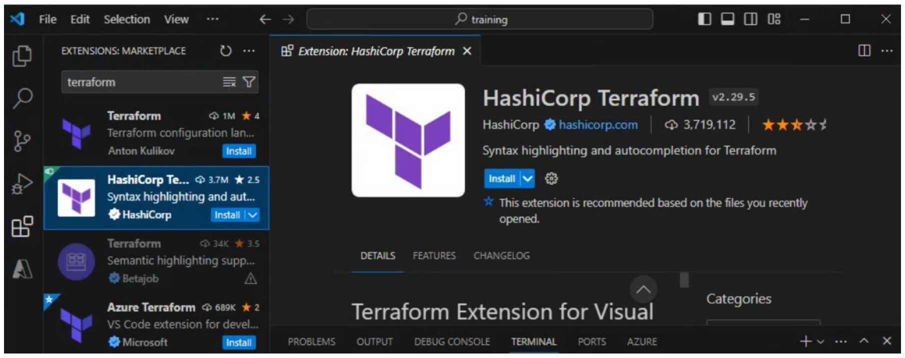
Azure CLI Tools v0.6.0

Microsoft microsoft.com | 1,647,664 | ★★★★☆

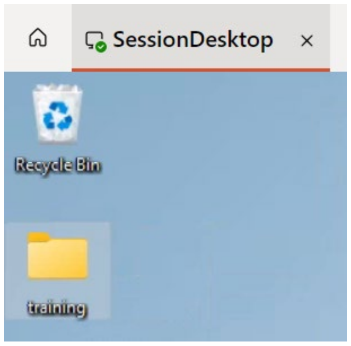
Tools for developing and running commands of the Azure CLI.

This extension is enabled globally.

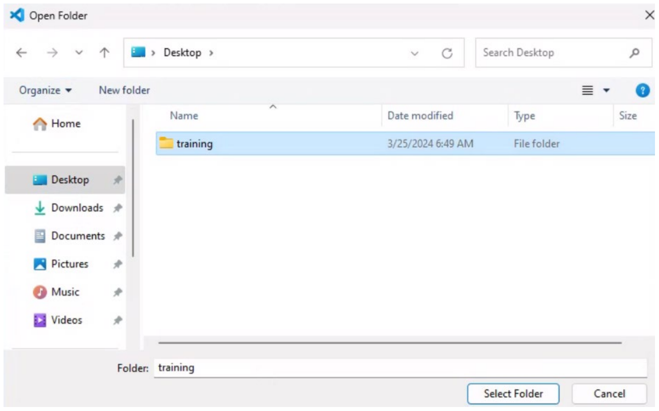
Azure CLI Tools


EXTENSIONS: MARKETPLACE
terraform
Terraform
Terraform configuration lan...
Anton Kulikov
Install
HashiCorp Te...
Syntax highlighting and aut...
HashiCorp
Install
Azure Terraform
VS Code extension for devel...
Microsoft
Extension: HashiCorp Terraform
HashiCorp Terraform v2.29.5
HashiCorp hashicorp.com 3,719,112
Syntax highlighting and autocompletion for Terraform
Install
This extension is recommended based on the files you recently opened.
DETAILS
FEATURES
CHANGELOG
Terraform Extension for Visual
Categories
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
AZURE

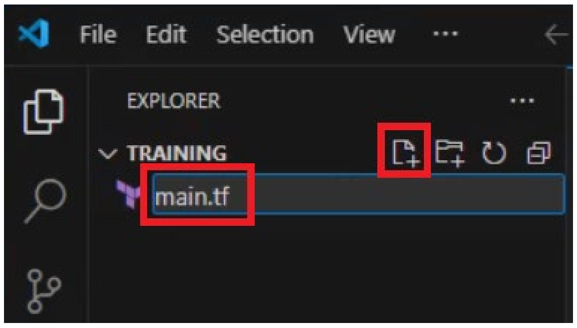
20. Create a folder on the desktop, named “training”.



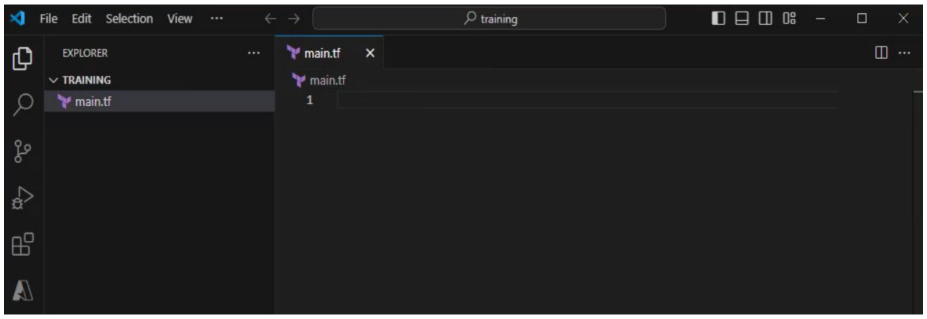
21. On the VS Code, click “Open Folder” and select the “training” folder.



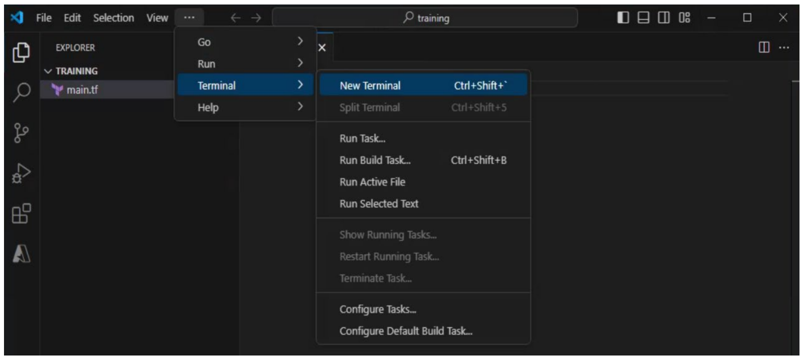
22. Create a new file, named “main.tf”, then click enter.



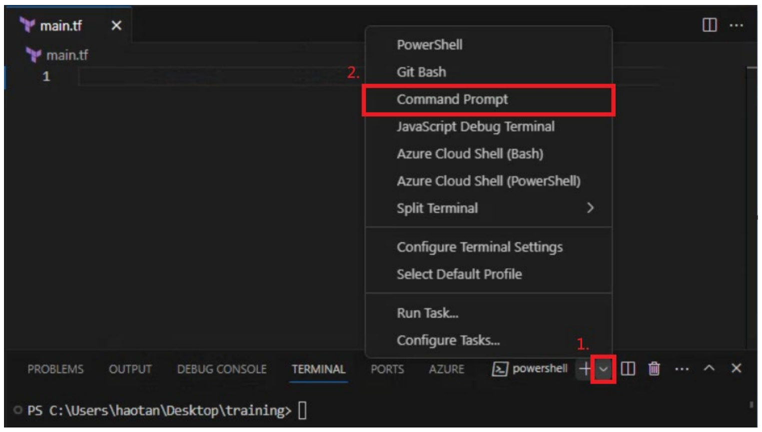
23. You can develop your Terraform now.



24. Click "•••" Icon -> Terminal -> New Terminal.

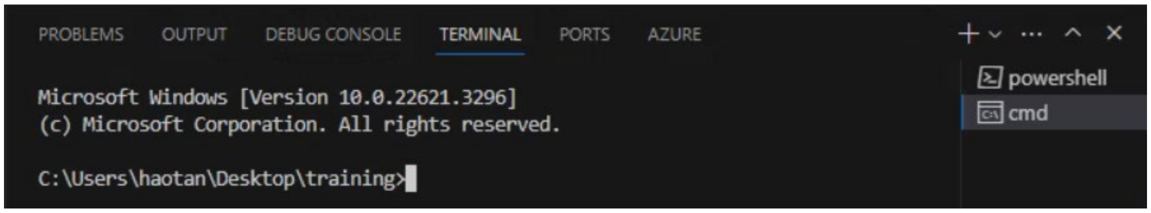


25. You can select the command line you prefer. In this case, we use "Command Prompt".



```
PS C:\Users\haotan\Desktop\training>
```

26. You can execute terraform relative command on the terminal now. And you can switch the terminal on the right-hand side.



```
Microsoft Windows [Version 10.0.22621.3296]
(c) Microsoft Corporation. All rights reserved.

C:\Users\haotan\Desktop\training>
```

Chapter 2: Terraform Configuration and workflow.

2.1 Terraform Configuration Files

The main way to write Terraform code is through a domain-specific language called HCL, or Hashicorp Configuration Language. By default, all files ending in a *.tf extension will be picked up by Terraform from the directory in which Terraform is run.

The two most fundamental building blocks to use Terraform are **providers** and **resources**. In our example, the provider block defines a way to interact with Azure, and the Azure provider has many different resource types available for provisioning. In this example, we provide a simple resource group on Azure.

```
provider "azurerm" {
  features {}
}


resource "azurerm_resource_group" "example" {
  name     = "example"
  location = "southeastasia"
}
```

2.2 Basic Terraform Use

Terraform (written in Go) is open source and distributed as binaries for many different platforms. It can be downloaded at https://www.terraform.io/downloads.html. The basic steps in the Terraform workflow are:

1. Write terraform code.
2. **terraform init**
   a. Automatically download any dependencies.
3. **terraform validate**
   a. Check Terraform configuration syntax and report errors.
4. **terraform plan**
   a. Refresh the state and compute what changes need to be made to reach the desired state in the Terraform configuration files.
5. **terraform apply**
   a. Perform the creation, alteration, and destruction of resources to reach the desired state.

2.3 Terraform Workflow Example

1. Open the Terminal or CMD on your laptop, execute **az login** command to login to your Azure account.
2. Execute cd command to change directories to the example directory.
3. Run **terraform init**. This will download any external dependencies, such as the “azure” provider. Terraform can automatically detect many of these and retrieve them seamlessly any time a **terraform init**.
4. Run **terraform validate**. This will ensure that there are no syntax errors in your Terraform script.

5. Run **terraform plan**. This will compare the state of resources on the provider with your local state. In this case, no infrastructure resources have been applied yet, so the plan will report adding a new resource, “azurerm_resource_group.example”, which is a resource group on Azure.
6. Run **terraform apply**. This will apply the changes from the plan.
7. Then we can check the resource on the Azure portal.

2.4 Providers

https://developer.hashicorp.com/terraform/language/providers

- Terraform is agnostic to the underlying platforms by supporting providers. A provider is responsible for understanding API interactions and exposing resources.
- There are many providers already created and ready for use out of the box:
  - Amazon Web Services (AWS)
  - Microsoft Azure
  - Google Cloud
  - VSphere
  - Etc...
- Terraform allows you to also write custom providers and there’s a huge community.
- Running terraform init downloads the needed plugins.

```
provider "azurerm" {

  features {}

}
```

2.5 Resources

https://developer.hashicorp.com/terraform/language/resources/syntax

- Resources are abstraction for anything that is configured and managed by Terraform.
- Within the block { } are the input arguments for the resource.
- Arguments (inputs) and attributes (can be thought of as outputs) are documented in the resource documentation.

```
resource "azurerm_resource_group" "example" {

  name     = "example"

  location = "southeastasia"

}
```

Lab 2: The Basic Terraform Workflow

In this lab, we will introduce the Terraform workflow using a simple example that creates a resource group on Azure. To do this, we'll need to use the two fundamental building blocks of Terraforms configuration language: providers and resources.

Providers are the interface to cloud providers such as AWS, GCP, Azure, and many others. They are configured with access credentials to provide Terraform with the ability to create, modify, and destroy infrastructure.

Resources are Terraforms fundamental interface to specifying what servers and services should be created, modified, or destroyed.

Core Terraform Workflow

In VS Code or File Explorer, create a directory for Lab2. To do this in VS Code, go to File -> Open Folder, and create a folder from the filesystem browser.

Create a Provider Block

Next, let's create a basic Terraform configuration with an Azure provider block. Create a new file in your VS Code project called main.tf.

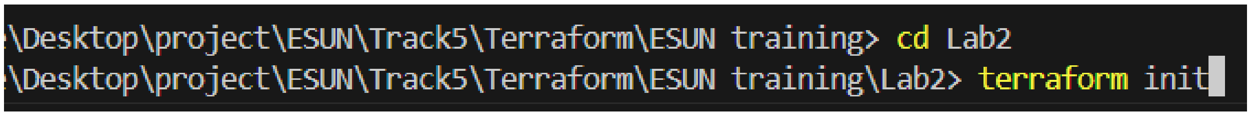
Insert the follow content into main.tf:

```
// This is an Azure provider block
provider "azurerm" {
  features {}
}
```

Save your file, and we're ready to run our first Terraform command.

Next, let's initialize our Terraform working directory. This should be done every time we start a new configuration or anytime, we check out an existing configuration. The command can be run over and over without changing the end state, a behavior known as idempotency.

In VS Code, open a terminal window with Terminal->New Terminal, or open a terminal for your operating system. Make sure that you're in the **Lab2** directory and type **terraform init**. This causes Terraform to download any plugins required to run our configuration, as well as create some supporting files that we'll learn more about later in the course.



```
\Desktop\project\ESUN\Track5\Terraform\ESUN training> cd Lab2
\Desktop\project\ESUN\Track5\Terraform\ESUN training\Lab2> terraform init
```

Hewlett Packard Enterprise

The typical output you can expect will look like this.

```
$ terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/azurerm...
- Installing hashicorp/azurerm v3.97.1...
- Installed hashicorp/azurerm v3.97.1 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the
provider selections it made above. Include this file in your version
control repository so that Terraform can guarantee to make the same
selections by default when you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform
plan" to see any changes that are required for your infrastructure.
All Terraform commands should now work.

If you ever set or change modules or backend configuration for
Terraform, rerun this command to reinitialize your working directory.
If you forget, other commands will detect it and remind you to do so
if necessary.
```

As you can see in the output, terraform suggests what our next step should be, which is **terraform plan**. The plan step of our workflow will generate a model of what the infrastructure we're creating through our configuration will look like after it is deployed. Let's run that now!

```
$ terraform plan

No changes. Your infrastructure matches the configuration.

Terraform has compared your real infrastructure against your
configuration and found no differences, so no changes are needed.
```

Creating a Resource Group using a Resource Block

Since we don't have any resources defined, our plan is empty. Now, let's add a Terraform resource block. The resource is Terraforms fundamental type that maps to some kind of infrastructure or service.

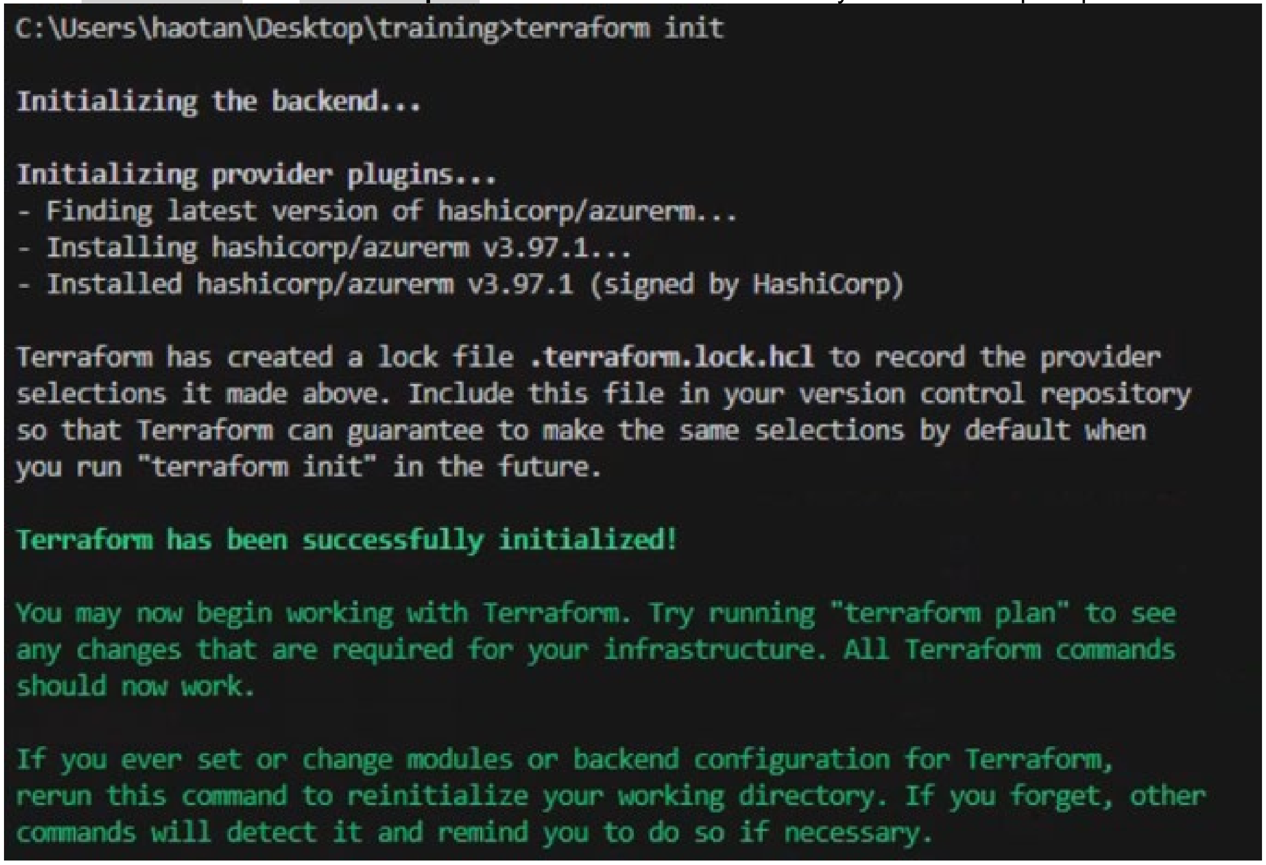
The `azurerm_resource_group` resource will create an Azure Resource Group. When it is created, the name and the location must be specified.

Hewlett Packard
Enterprise

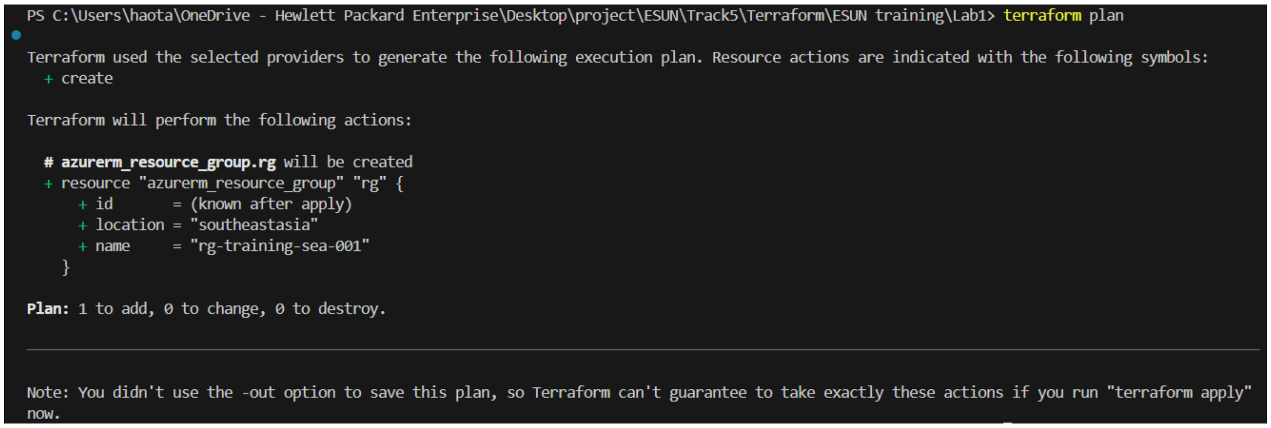
Now modify your main.tf file to include the following resource block to create a resource of type `azurerm_resource_group` named `rg`. Then replace the name of the resource group with your **employee ID** and **SAVE** the file.

```
// This is an Azure provider block
provider "azurerm" {
  features {}
}

// This is an Azure resource block that creates a resource group
resource "azurerm_resource_group" "rg" {
  name     = "rg-training-sea-<employee_id>"
  location = "southeastasia"
}
```

Run a **terraform init** and **terraform plan** from the VS Code terminal or your command prompt window.



```
C:\Users\haotan\Desktop\training>terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/azurerm...
- Installing hashicorp/azurerm v3.97.1...
- Installed hashicorp/azurerm v3.97.1 (signed by HashiCorp)

Terraform has created a lock file .terraform.lock.hcl to record the provider
selections it made above. Include this file in your version control repository
so that Terraform can guarantee to make the same selections by default when
you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
```

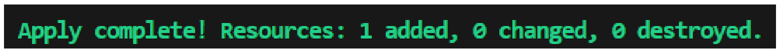


```
PS C:\Users\haota\OneDrive - Hewlett Packard Enterprise\Desktop\project\ESUN\Track5\Terraform\ESUN training\Lab1> terraform plan

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 = "southeastasia"
      + name     = "rg-training-sea-001"
    }

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.
```

As you can now see, our plan output is a little more useful. It shows that terraform will attempt to add 1 new resource, a resource group(`azurerm_resource_group`), by the internal reference name of rg.

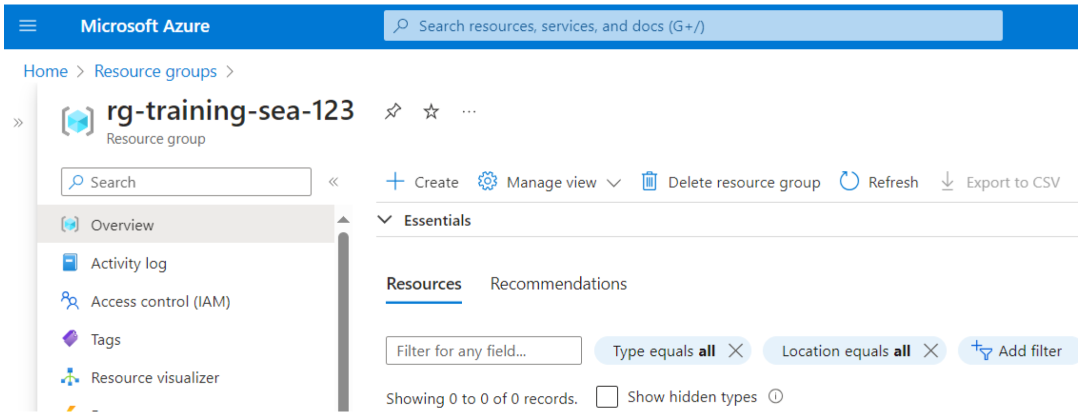
There are 3 attributes for the `azurerm_resource_group` resource type: id, location, and name. If you wanted to change any of the other items in the plan list you could reference the azurerm_resource_group in Terraform documentation and add the appropriate components to your resource definition.
https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/resource_group

Then please run the **terraform apply** command to create the resource group.



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

Check if the resource group was created on the Azure portal.



rg-training-sea-123
Resource group

Showing 0 to 0 of 0 records.

Creating a Virtual Network using a Resource Block

Let's add another Terraform resource block to create a vNet. In the main.tf file, replace the name of **employee ID**. You can find more details about the resource azurerm_virtual_network on the documentation:
https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/resources/virtual_network

```
// This is an Azure provider block
provider "azurerm" {
  features {}
}

// This is an Azure resource block that creates a resource group
resource "azurerm_resource_group" "rg" {
  name     = "rg-training-sea-<employee_id>"
  location = "southeastasia"
}

// This is an Azure resource block that creates a virtual network
resource "azurerm_virtual_network" "vnet" {
  name                = "vnet-training-sea-<employee_id>"
  location            = "southeastasia"
  resource_group_name = "rg-training-sea-<employee_id>"
  address_space       = ["10.0.0.0/16"]
  tags = {
    environment = "Test"
  }
  depends_on = [ azurerm_resource_group.rg ]
}
```

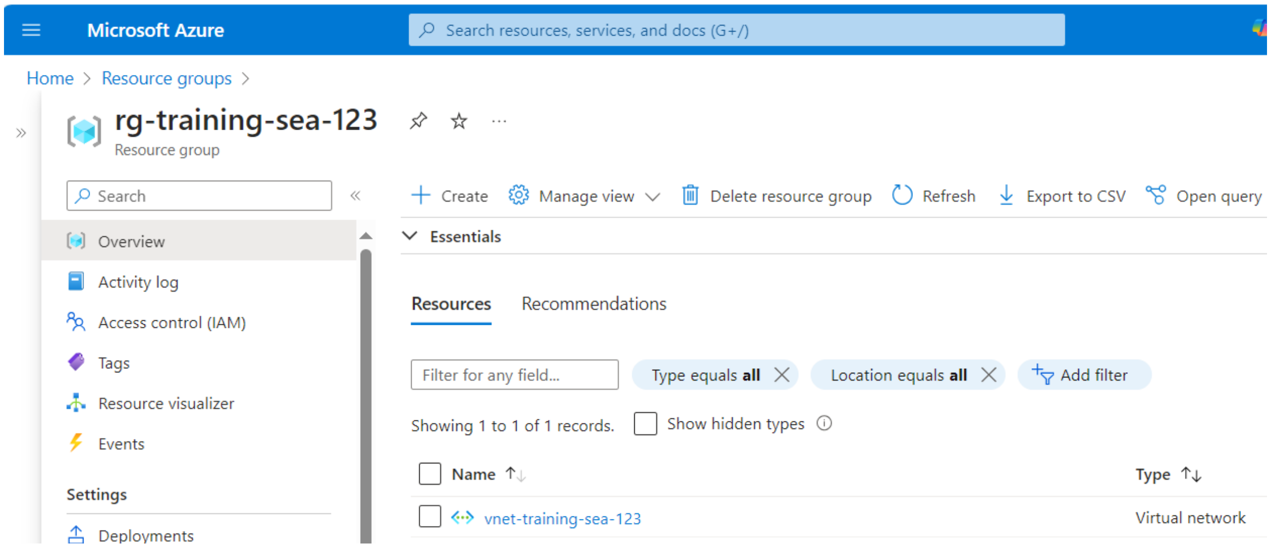
Run a **terraform init** and **terraform plan** from the VS Code terminal or your command prompt window.

```
Plan: 2 to add, 0 to change, 0 to destroy.
```

Please run the **terraform apply** command to create the resource group and vnet.

```
Apply complete! Resources: 2 added, 0 changed, 0 destroyed.
```

Check if the resource group and vNet created on the Azure portal.



Showing 1 to 1 of 1 records.

Name	Type
vnet-training-sea-123	Virtual network