**Terraform Basics**

Terraform is an **Infrastructure as Code (IaC)** tool developed by **HashiCorp** that allows users to define and manage infrastructure using a declarative configuration language HCL (HashiCorp configuration language).

It enables automation of provisioning, updating, and scaling infrastructure across cloud providers like AWS, Azure, and Google Cloud.

**Key Features of Terraform:**

1. **Declarative Configuration**

Uses **HashiCorp Configuration Language (HCL)** to define infrastructure in a human-readable format.

1. **Multi-Cloud Support**

Works with **AWS, Azure, Google Cloud, Kubernetes, and on premise solutions**.

1. **Infrastructure as Code (IaC)**

Infrastructure is defined in code, making it version-controlled and reusable.

1. **Immutable Infrastructure**

Changes replace existing resources rather than modifying them in place, reducing configuration drift. (If we want to update any server or data base. it will create entire new server with desired changes. Instead of making changes to old server).

1. **State Management**

Maintains the infrastructure state file (terraform.tfstate) to track resource changes.

1. **Modular and Reusable**

Supports **modules** to create reusable infrastructure components.

1. **Dependency Management**

Automatically determines resource dependencies and provisions them in the correct order.

1. **Execution Planning (**terraform plan**)**

Shows a preview of changes before applying them.

1. **Automated Provisioning and Scaling**

Can be used with CI/CD pipelines for automated deployments.

1. **Rollback and Change Management**

Uses **Version Control** (Git) for rollback and tracking changes.

### ****NOTE: Ide potency in Terraform:****

**Ide potency** is a key feature of Terraform that ensures running the same configuration multiple times results in the **same state** of infrastructure, without unintended changes. This means:

**That means** applying the same Terraform configuration multiple times won’t cause duplicate resources or unintended modifications.

(Running the same terraform scrip multiple times, it won’t change the present infrastructure).

**NOTE: Splitting Terraform Files**:

We can split Terraform files into multiple files to improve organization and maintainability.

Terraform will automatically merge all .tf files in a directory when executing commands.

### ****Main Configuration Files****

* main.tf → contains primary configurations like providers, backend, and resource definitions.
* variables.tf → defines input variables.
* outputs.tf →Defines output values.
* terraform.tfvars → Provides values for variables.
* provider.tf → Specifies provider configurations (e.g., AWS, Azure).

### ****Breaking down Resources****

You can create separate files for different resources, such as:

* network.tf → VPC, subnets, security groups.
* compute.tf → EC2 instances, Kubernetes clusters.
* storage.tf → S3 buckets, databases.
* iam.tf → IAM roles, policies.

**Advantages:**

1. Infrastructure as a code (IAC).
2. Version control.
3. Automation
4. Support multi-cloud.
5. Declarative syntax.
6. State management.
7. Modularity & reusability.

**Disadvantages:**

1. Managing state file is challenging.
2. It depends on some third-party plugins.
3. Debugging is complex when dealing with complex dependences.
4. Lack of terraform support.
5. Newly launched services cannot manages by terraform immediately.

Terraform configuration file has two main blocks:

1. Provider Blocker.
2. Resource Block.

**Providers** {

AZURE or

GCP or

AWS

}

**Resource** {

RG or

VM or

Vnet and so on…

}

In order to execute the terraform code we have to use some terraform commands they are:

**terraform init**  🡺it initialize the terraform working directory by downloading required provider plugins and also it can configures the Backend and prepares the modules.

**terraform validate**🡺 it is used to check the **syntax** and **validity** of a Terraform configuration file before applying any changes.

**terraform plan**🡺it is used to preview the changes Terraform will make to your infrastructure before applying them.

**terraform apply**🡺 it is used to **execute the changes** described in the Terraform configuration.

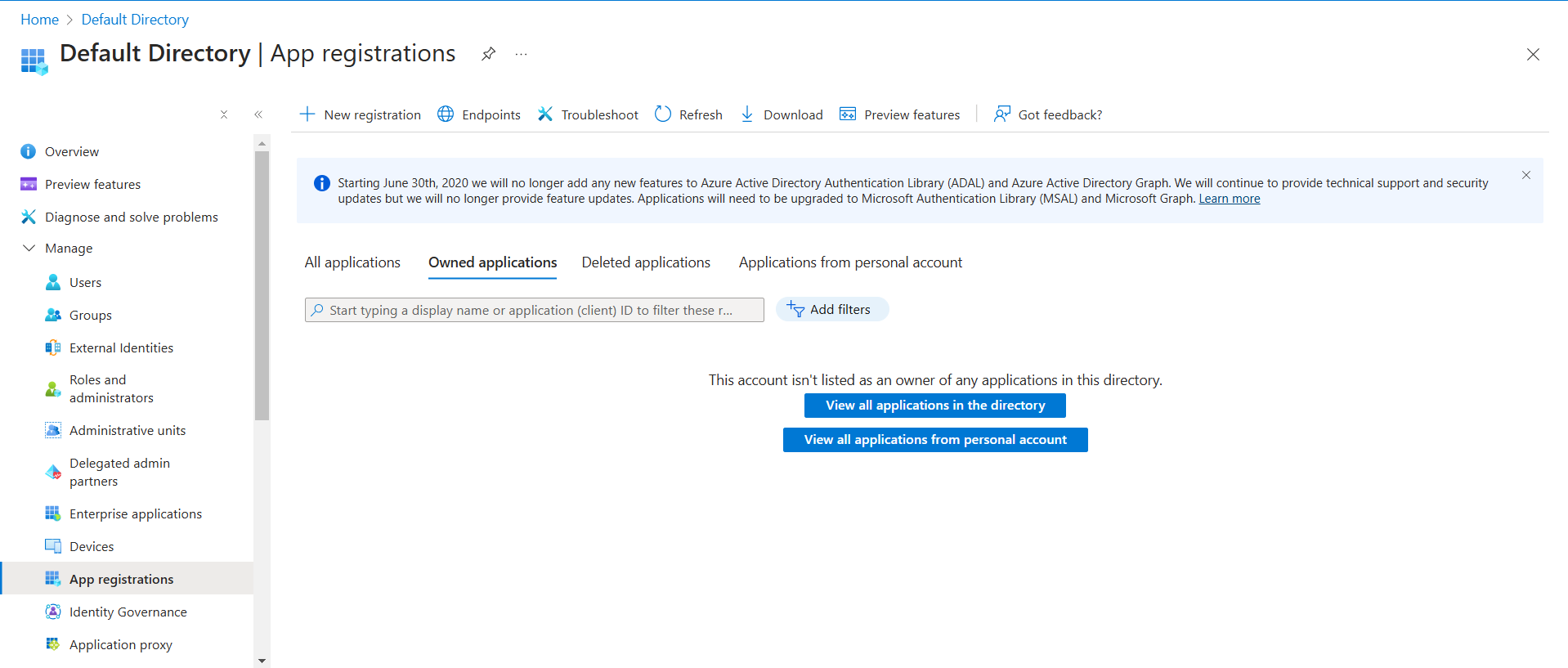
Let’s perform the task to create the resource group in azure using the terraform command.

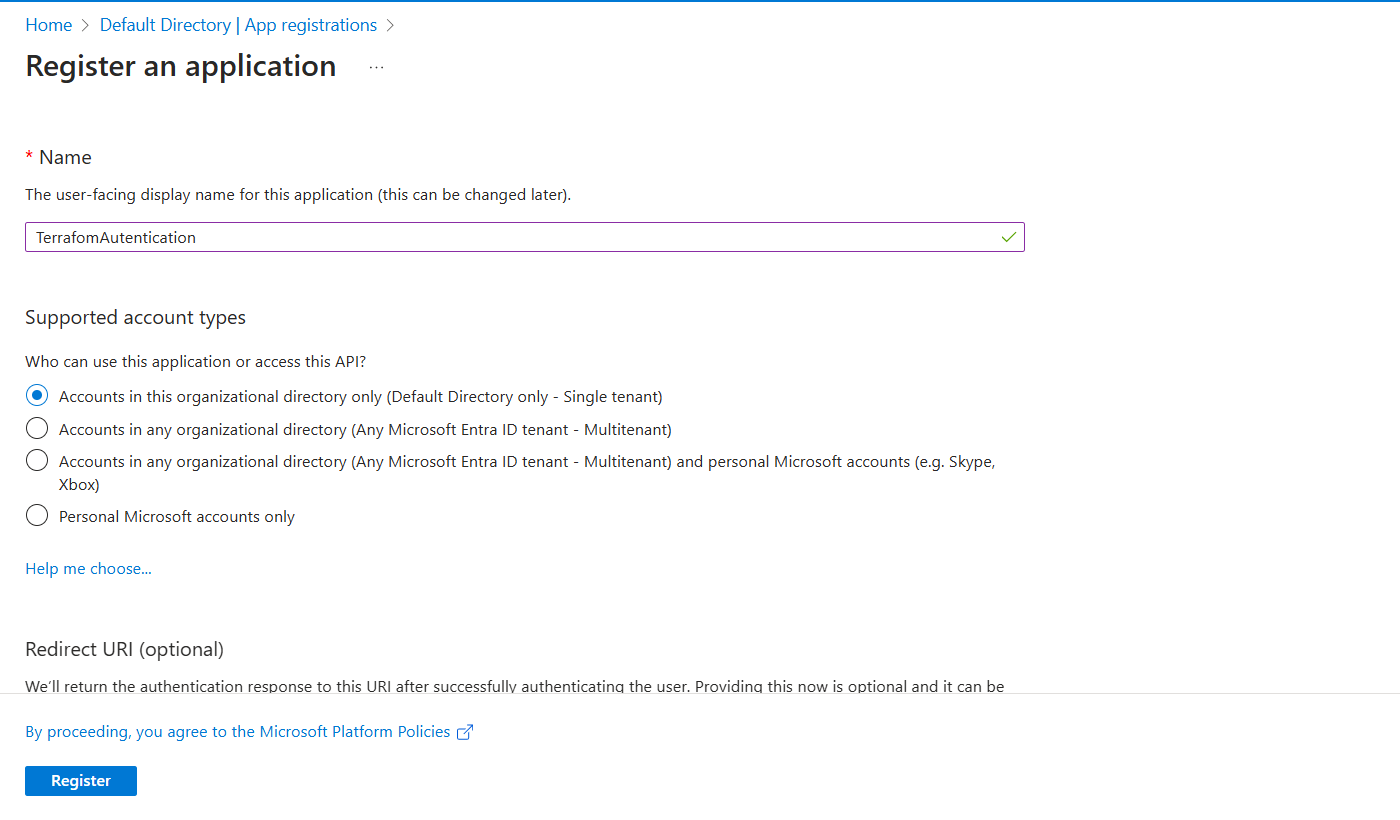
**step1:** Configure the provider block in main.tf file

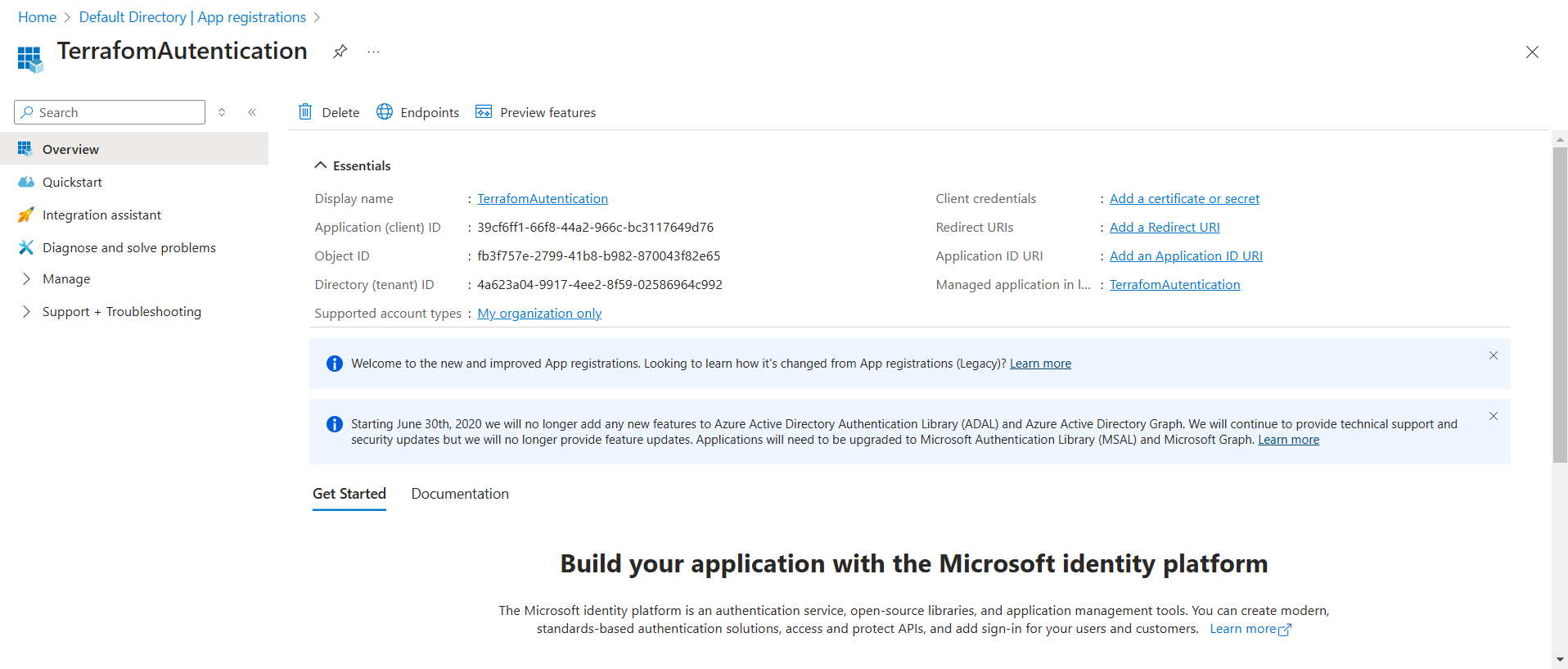
We know that the terraform is a third part tool so in order to connect with azure portal we need the authentication method.

For authentication we use the **service principle account.**

**Login into the azure portal🡺 search for entraid 🡺app registration🡺 new registration**

****



**Fig:** App registration with name “TerraformAuthentication”

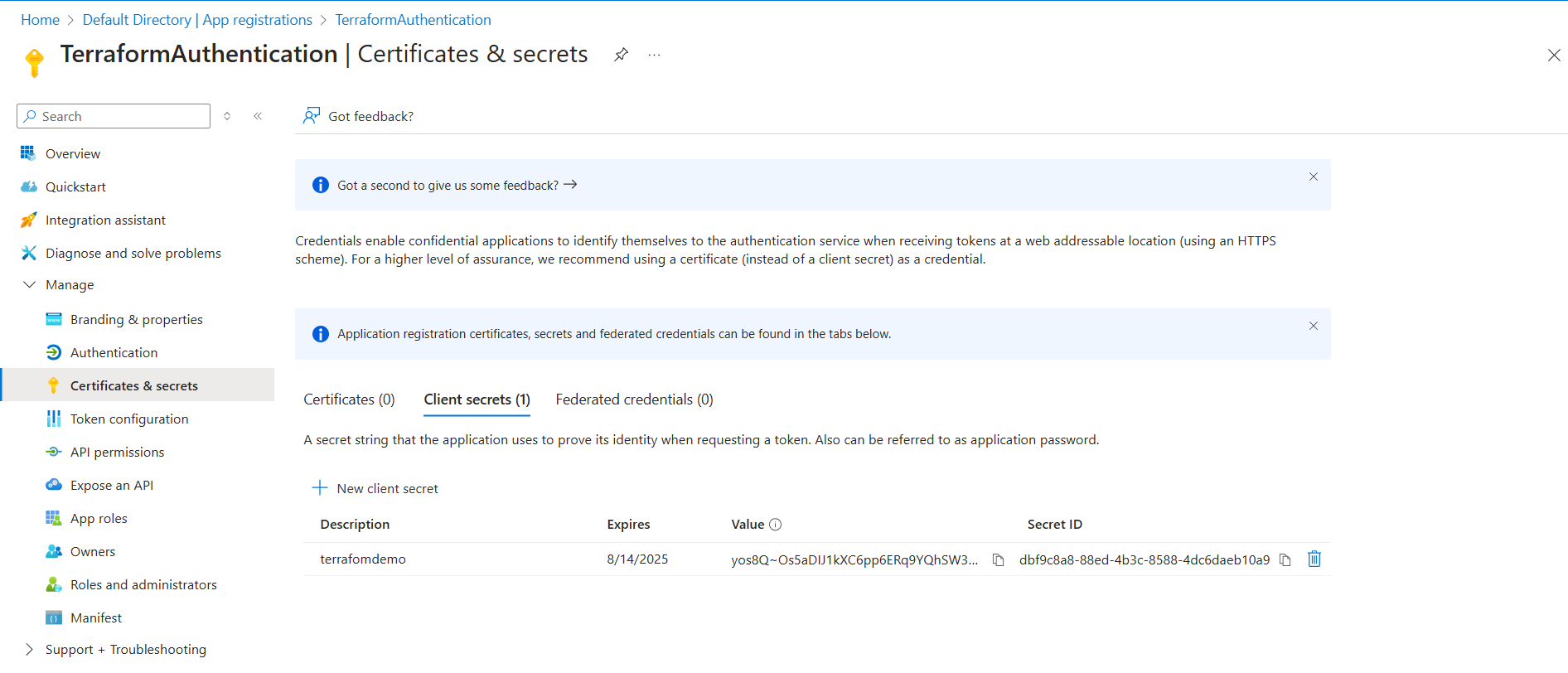


Fig: creating of client secrete key value.

**Note:** Assign the Owner role to this App registrar (TerraformAuthentication).

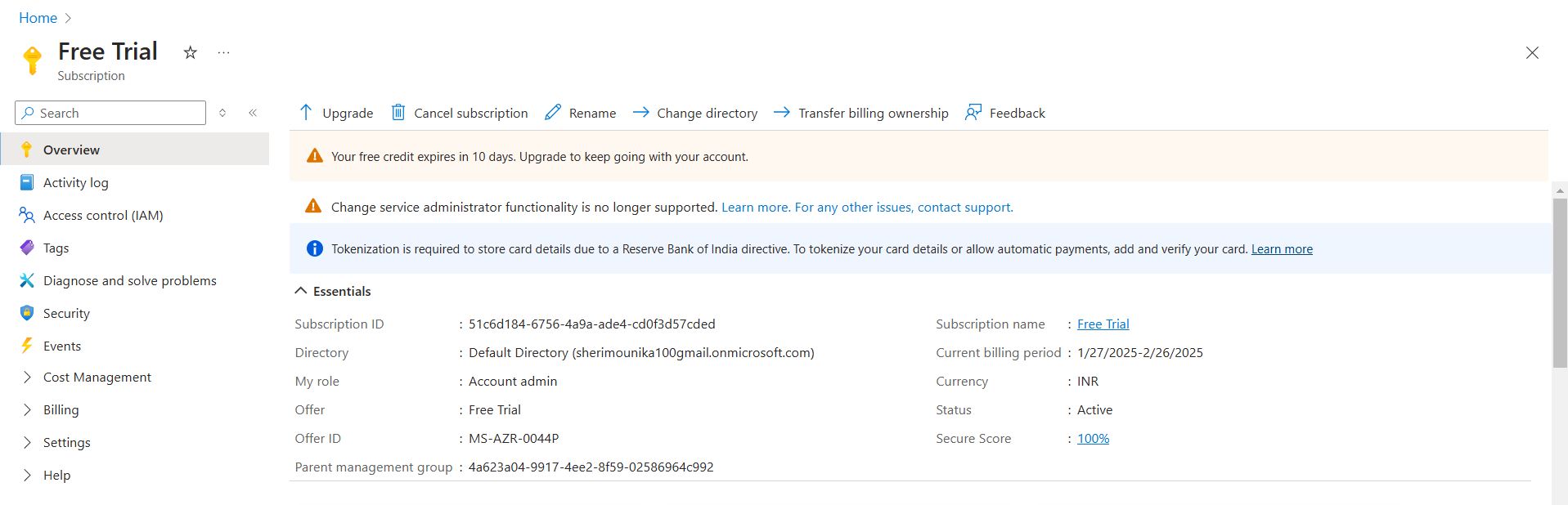


Fig: Azure subscription ID



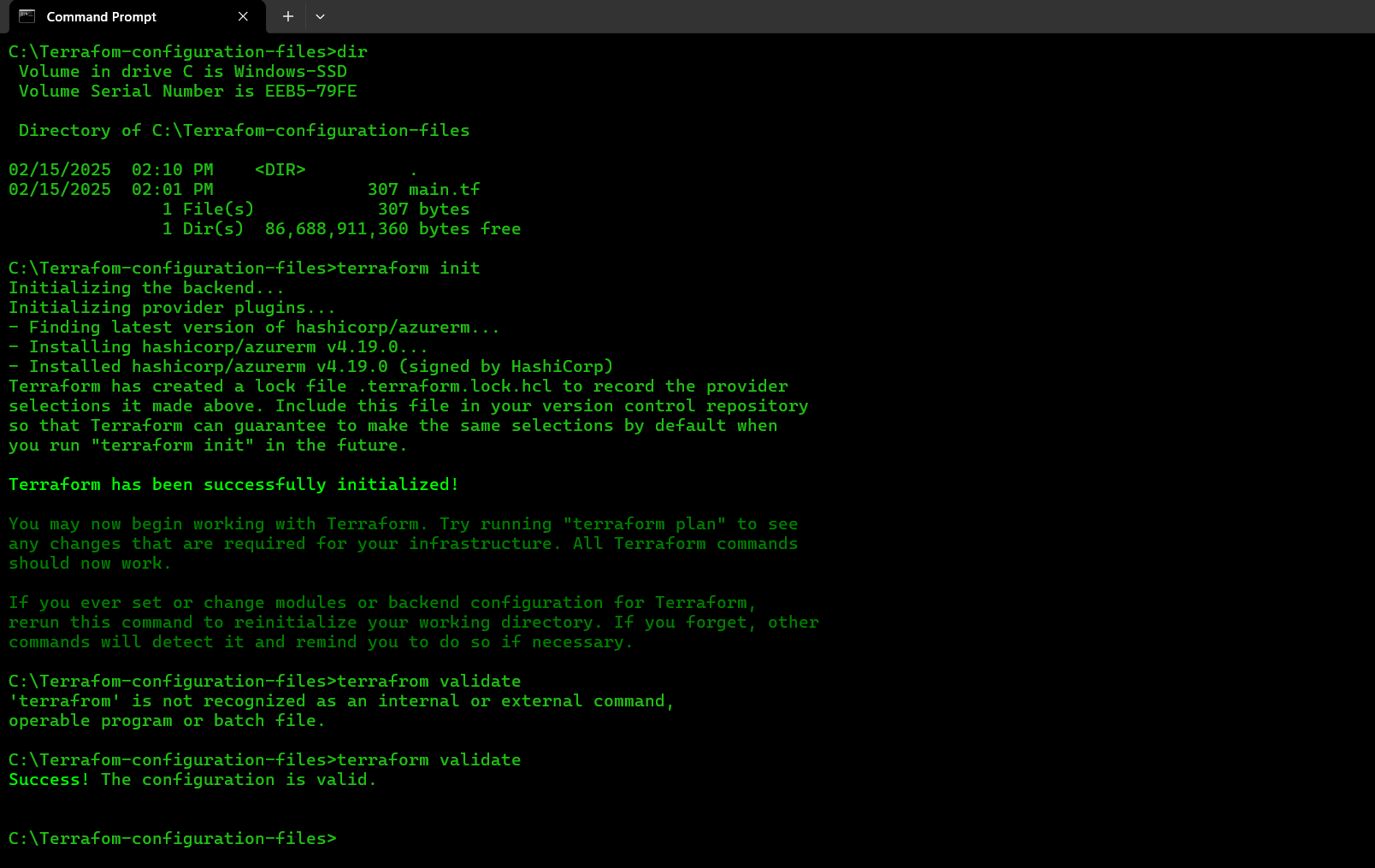
# This is provider block. Authenticating using a Service Principal with a Client Secret

This is optional block. If we declared or not manually it will by default takes the current version of azure plugin

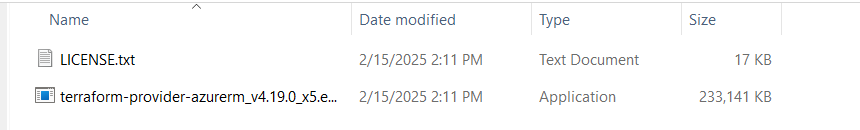
Step2: Create the resource block in the terraform main.tf file.



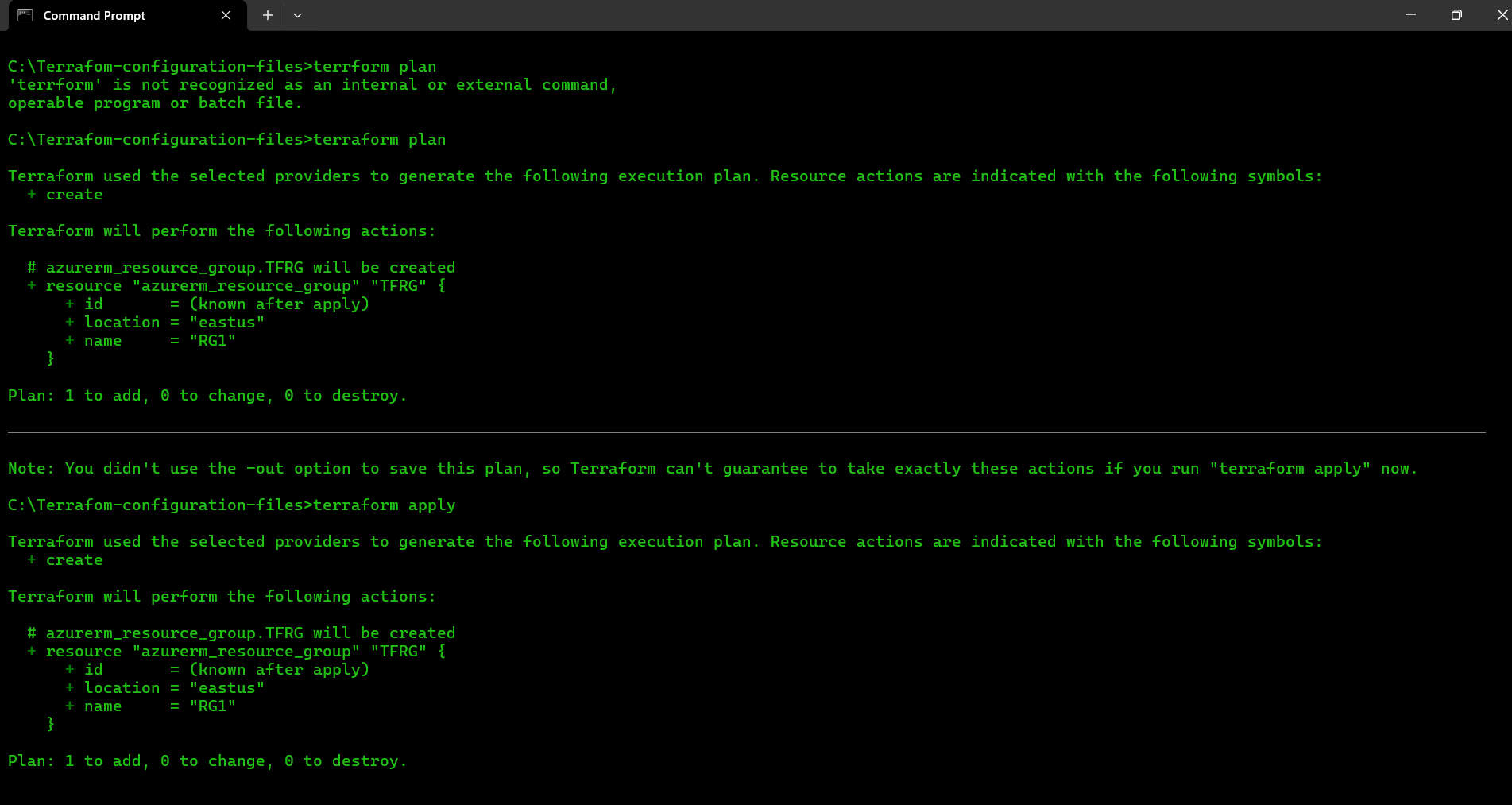
Step3: Let’s execute the terraform main.tf file by entering into the main.tf file path.



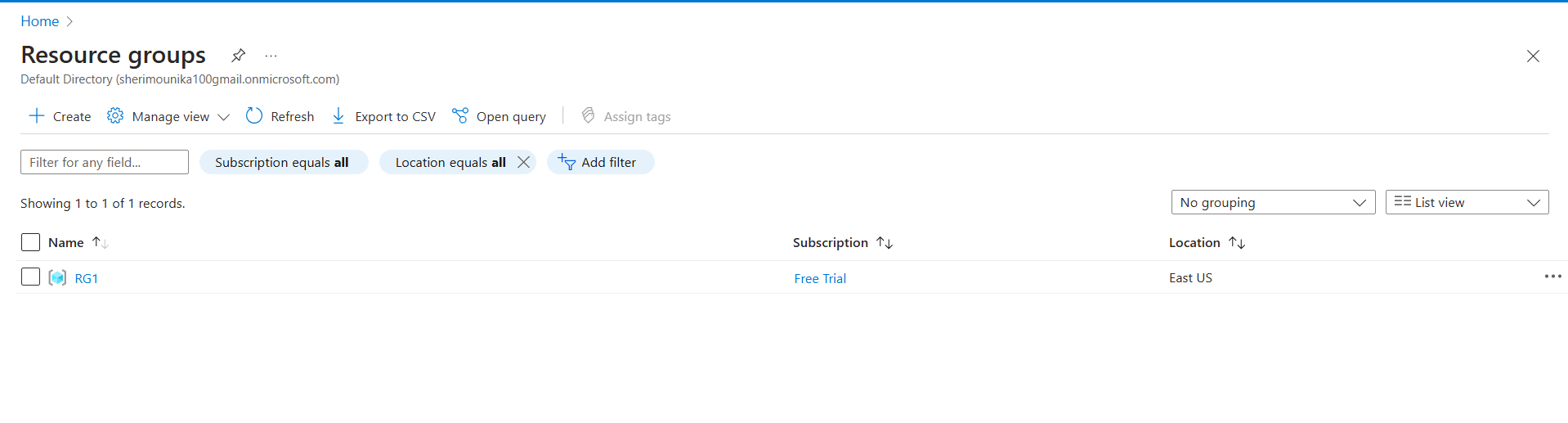
**Note:** when we execute the command “terraform init” it will download the azure plugin’s as shown below figure.



Step5: Now run the terraform code by using the command terraform apply.



Step6: Let’s check in the portal whether it is created or not.



Step7: Now destroy the resources which are created in the portal using command “terraform destroy”

