

Project Name:

SimpleTimeService-Docker-K8s-Terraform Using Azure Cloud

Table of Contents

1. Project Overview
2. Pre-requisites
3. Create a Simple Application
4. Dockerizing the Application
5. Infrastructure as Code (Terraform on Azure)
6. Deployment to Azure Kubernetes Service

1. Project Overview:

SimpleTimeService is a microservice that provides the current timestamp and the IP address. It is designed for minimalist deployment using Docker and Kubernetes.

This project also includes Terraform scripts to automate the deployment of the service on AZURE using AKS.

2. Pre-requisites:

- Docker
- Python
- Azure Cloud (AKS, VNET, RESOURCE GROUP)
- Terraform
- Azure CLI

3. Create a Simple Application:

After Completion of Python setup. Run the **pip install flask** because flask is a web framework to creates a Webservice.

```
SimpleTimeService.py X
C: > Users > ktssk > Downloads > SimpleTimeService > SimpleTimeService.py
1  from flask import Flask, jsonify, request
2  from datetime import datetime
3
4  app = Flask(__name__)
5
6  @app.route("/")
7  def home():
8      return jsonify({
9          "timestamp": datetime.utcnow().isoformat(),
10         "ip": request.remote_addr
11     })
12
13 if __name__ == "__main__":
14     app.run()
15
```

Description of the Code:

```
from flask import Flask, jsonify, request
```

```
from datetime import datetime
```

- **Flask:** A Python web framework used to create this web service.
- **jsonify:** Converts Python dictionaries into JSON responses.
- **request:** Extract details from incoming HTTP requests
- **datetime:** Used to fetch the current date and time.

```
app = Flask(__name__)
```

- This creates an instance of the **Flask** web application.

```
@app.route("/")
```

```
def home():
```

```
    return jsonify({
        "timestamp": datetime.utcnow().isoformat(),
        "ip": request.remote_addr
    })
```

- `@app.route("/")`: Defines a **route** for the root URL (/), meaning when users visit the server's page.

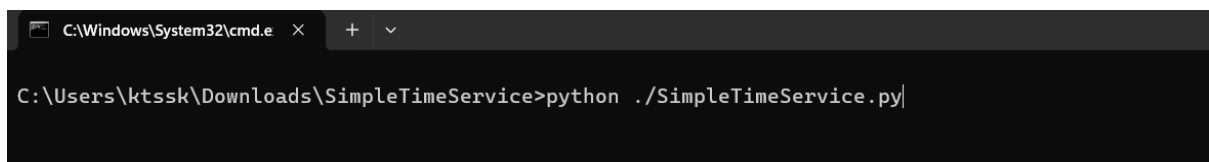
- `datetime.utcnow().isoformat()`: Gets the current **UTC time** in **ISO 8601 format**.
- `request.remote_addr`: Extracts the **IP address** of the making the request.
- `jsonify(...)`: Converts the response to **JSON format**.

```
if __name__ == "__main__":
```

```
    app.run()
```

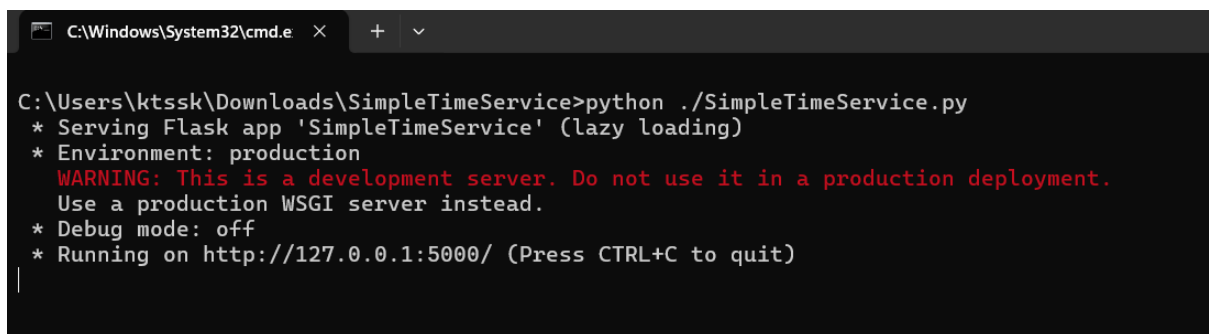
- **if __name__ == "__main__":** ensures that the script runs only if executed directly.
- **app.run()** starts the Flask web server.

→ Run the Python code in the terminal.



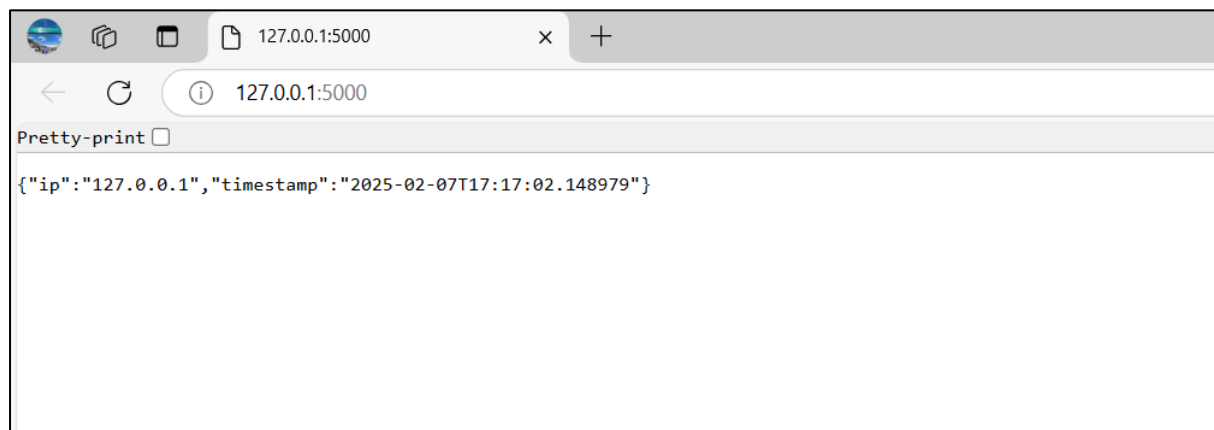
```
C:\Windows\System32\cmd.e  x  +  v
C:\Users\ktssk\Downloads\SimpleTimeService>python ./SimpleTimeService.py
```

→ Output is the response includes the client's IP address.



```
C:\Users\ktssk\Downloads\SimpleTimeService>python ./SimpleTimeService.py
* Serving Flask app 'SimpleTimeService' (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

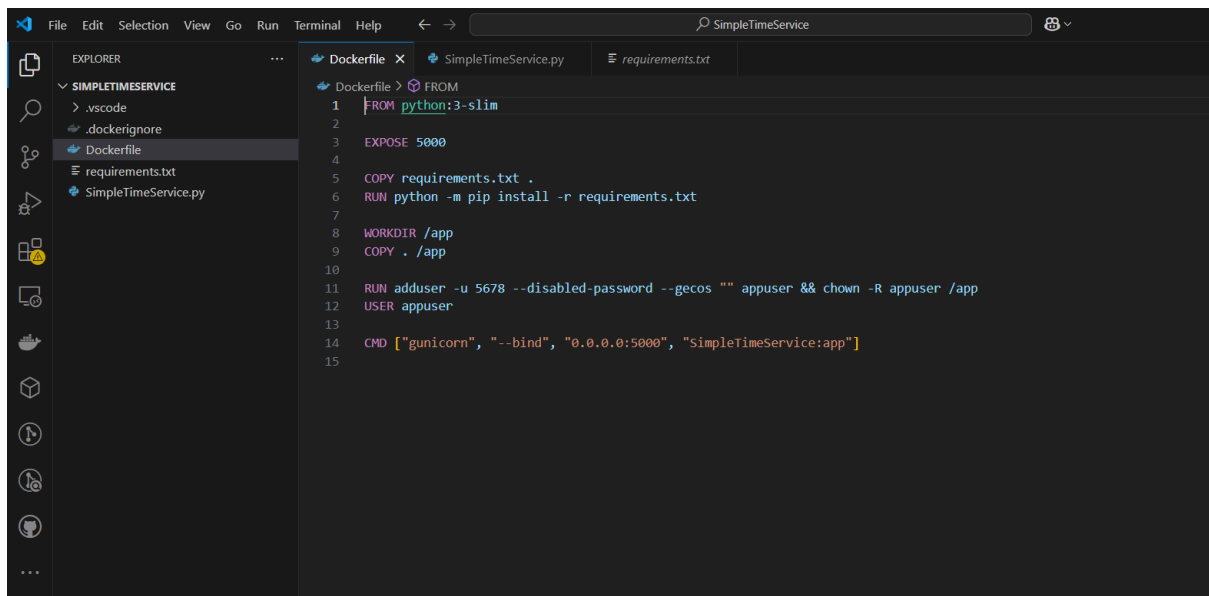
→ Browse the Localhost URL to view the Application Webpage.



4. Dockerizing the Application:

→ Create a Docker file in the same Code Application Folder to load the code dependencies to the docker file.

→ Write a Docker file based on the code.

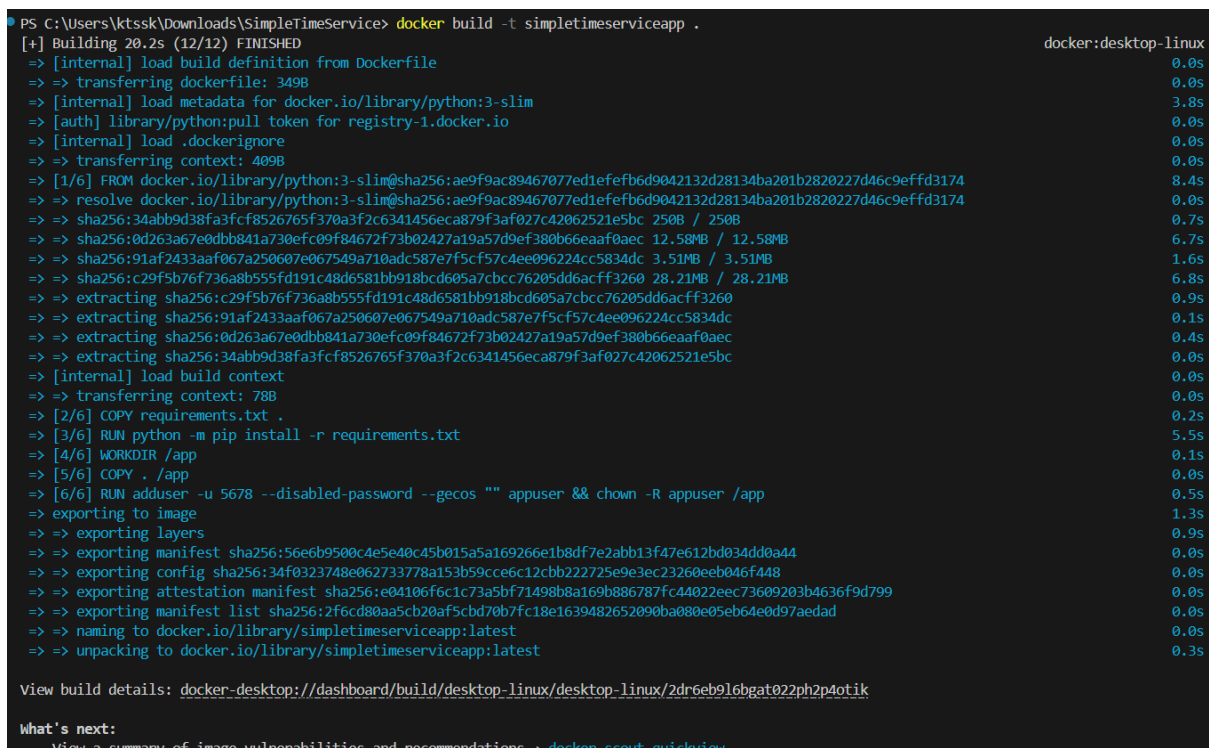


The screenshot shows the VS Code editor with a Dockerfile open. The Explorer sidebar on the left shows the project structure: SIMPLETIMESERVICE, .vscode, .dockerignore, Dockerfile, requirements.txt, and SimpleTimeService.py. The Dockerfile content is as follows:

```
Dockerfile > FROM
1 FROM python:3-slim
2
3 EXPOSE 5000
4
5 COPY requirements.txt .
6 RUN python -m pip install -r requirements.txt
7
8 WORKDIR /app
9 COPY . /app
10
11 RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser /app
12 USER appuser
13
14 CMD ["gunicorn", "--bind", "0.0.0.0:5000", "SimpleTimeService:app"]
15
```

→ Build the Docker Image with Docker build command.

docker build -t simpletimeserviceapp .



The screenshot shows a terminal window with the command `docker build -t simpletimeserviceapp .` and its output. The output shows the progress of building the Docker image, including the steps of loading build definition, transferring context, resolving dependencies, and exporting the image. The final output is:

```
PS C:\Users\ktssk\Downloads\SimpleTimeService> docker build -t simpletimeserviceapp .
[+] Building 20.2s (12/12) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 349B
=> [internal] load metadata for docker.io/library/python:3-slim
=> [auth] library/python:pull token for registry-1.docker.io
=> [internal] load .dockerignore
=> => transferring context: 409B
=> [1/6] FROM docker.io/library/python:3-slim@sha256:ae9f9ac89467077ed1efefb6d9042132d28134ba201b2820227d46c9effd3174
=> => resolve docker.io/library/python:3-slim@sha256:ae9f9ac89467077ed1efefb6d9042132d28134ba201b2820227d46c9effd3174
=> => sha256:34abb9d38fa3fcf8526765f370a3f2c6341456eca879f3af027c42062521e5bc 250B / 250B
=> => sha256:0d263a67e0dbb841a730efc09f84672f73b02427a19a57d9ef380b666eaaaf0aec 12.58MB / 12.58MB
=> => sha256:91af2433aaf067a250607e067549a710adc587e7f5cf57c4ee096224cc5834dc 3.51MB / 3.51MB
=> => sha256:c29f5b76f736a8b555fd191c48d6581bb918bcd605a7cbcc76205dd6acff3260 28.21MB / 28.21MB
=> => extracting sha256:c29f5b76f736a8b555fd191c48d6581bb918bcd605a7cbcc76205dd6acff3260 0.9s
=> => extracting sha256:91af2433aaf067a250607e067549a710adc587e7f5cf57c4ee096224cc5834dc 0.1s
=> => extracting sha256:0d263a67e0dbb841a730efc09f84672f73b02427a19a57d9ef380b666eaaaf0aec 0.4s
=> => extracting sha256:34abb9d38fa3fcf8526765f370a3f2c6341456eca879f3af027c42062521e5bc 0.0s
=> [internal] load build context
=> => transferring context: 78B
=> [2/6] COPY requirements.txt .
=> [3/6] RUN python -m pip install -r requirements.txt
=> [4/6] WORKDIR /app
=> [5/6] COPY . /app
=> [6/6] RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -R appuser /app
=> exporting to image
=> => exporting layers
=> => exporting manifest sha256:56e6b9500c4e5e40c45b015a5a169266e1b8df7e2abb13f47e612bd034dd0a44
=> => exporting config sha256:34f0323748e062733778a153b59cce6c12cbb222725e9e3ec23260eeb046f448
=> => exporting attestation manifest sha256:e04106f6c1c73a5bf71498b8a169b886787fc44022eec73609203b4636f9d799
=> => exporting manifest list sha256:2f6cd80aa5cb20af5cbd70b7fc18e1639482652090ba080e05eb64e0d97aead
=> => naming to docker.io/library/simpletimeserviceapp:latest
=> => unpacking to docker.io/library/simpletimeserviceapp:latest
0.3s

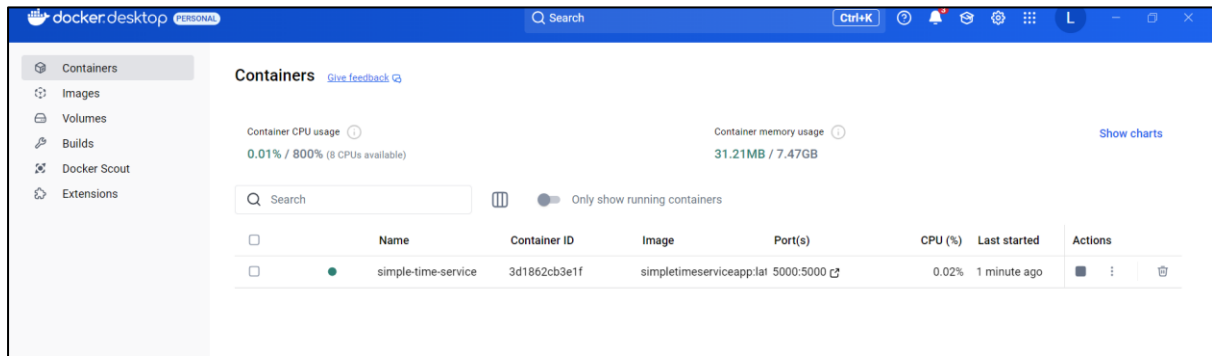
view build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/2dr6eb916bgat022ph2p4otik

What's next:
View a summary of image vulnerabilities and recommendations → docker scout quickview
```

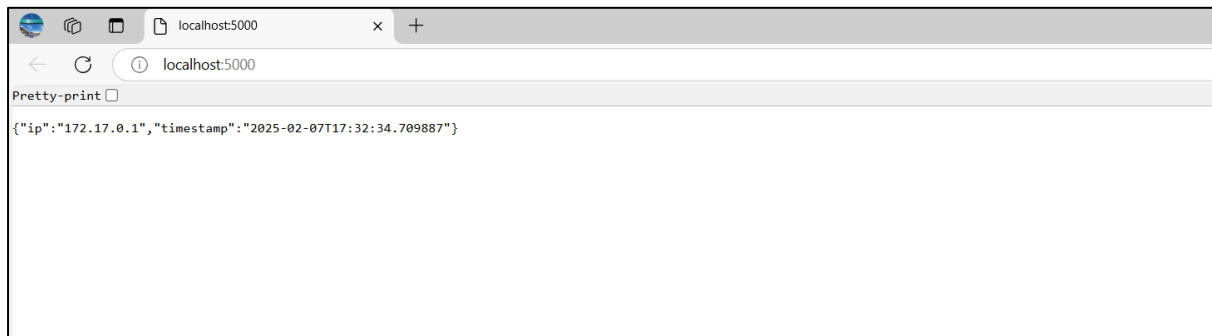
→Run the image to Create a Container

```
PS C:\Users\ktssk\Downloads\SimpleTimeService> docker run -d -p 5000:5000 --name simple-time-service simpletimeserviceapp:latest 3d1862cb3e1fcbed033e699082b771b4d5b15a071dd77d52bf4f3269cd77f45d
```

→Go to Docker Engine and verify the container.

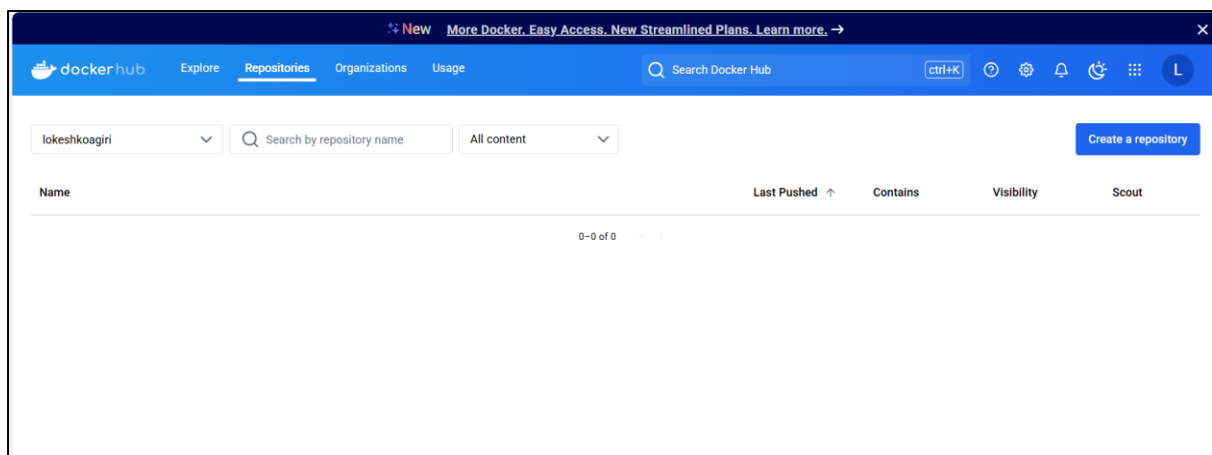


→Open the Located port to view.

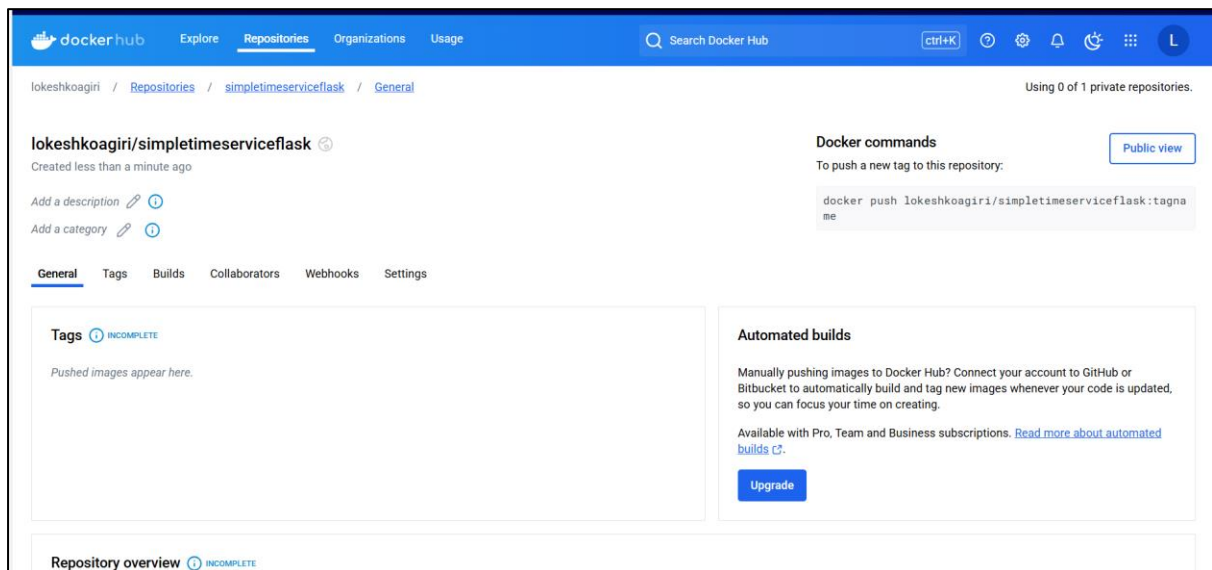


→Push the Docker image to the Docker Hub. Go to hub.docker.com and login with your Credentials to store our docker image to Docker hub.

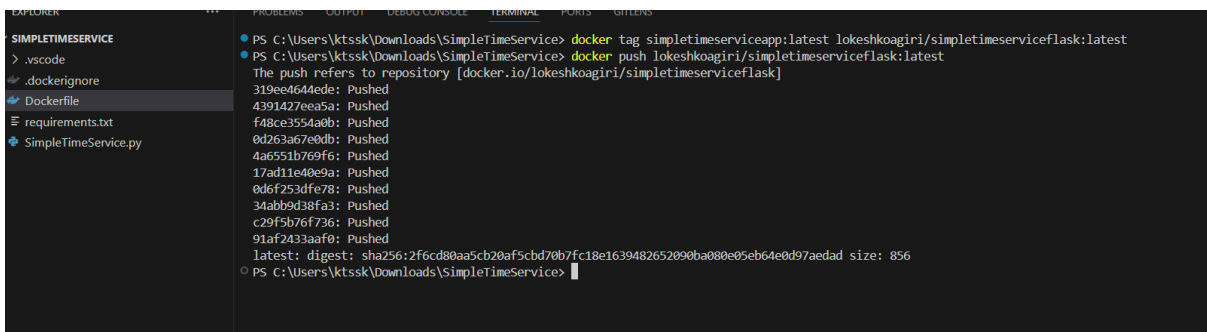
→Create a Repository and push the image to hub.



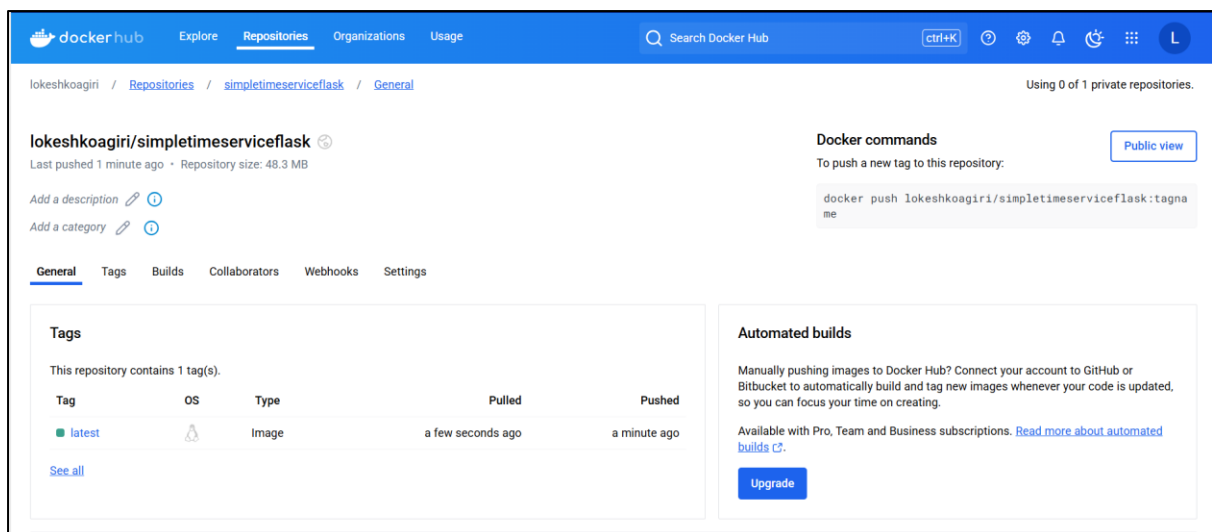
→ After created the Repository look like this.



→ Tag and push the Docker image to docker hub with commands.

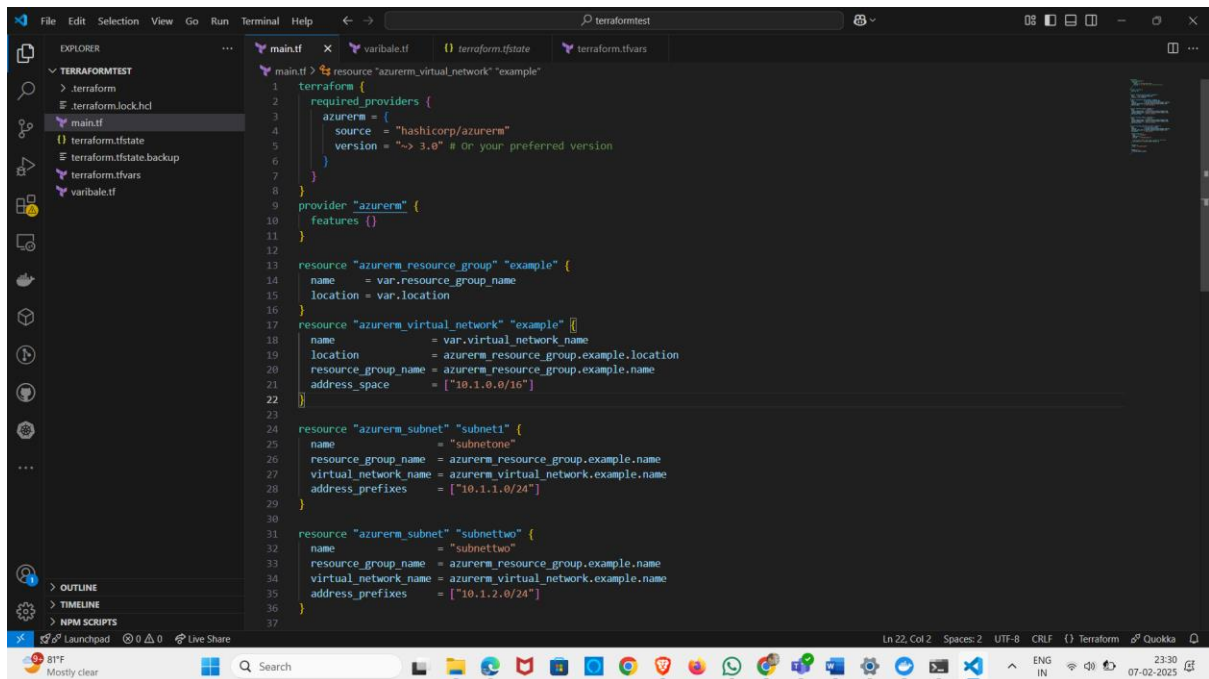


→ Verify the image is stored in the Docker HUB.



5. Infrastructure as Code (Terraform on Azure):

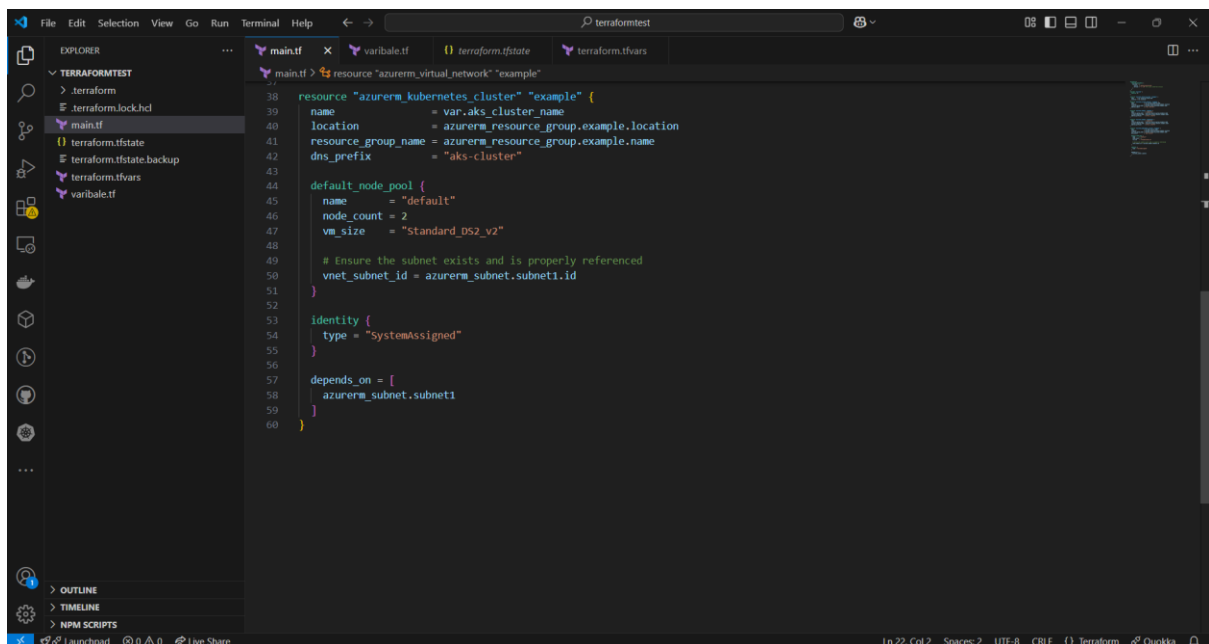
→ Create a Terraform script with main.tf



This screenshot shows a Visual Studio Code editor with a Terraform script named `main.tf` open. The script defines the following resources:

- `terraform`: The main configuration block.
- `required_providers`: Specifies the `azurerm` provider from `hashicorp/azurerm` with a version constraint of `>= 3.0`.
- `provider "azurerm"`: The provider configuration block.
- `resource "azurerm_resource_group" "example"`: A resource group named `example` in the location `var.location`.
- `resource "azurerm_virtual_network" "example"`: A virtual network named `example` with address space `["10.1.0.0/16"]`.
- `resource "azurerm_subnet" "subnet1"`: A subnet named `subnet1` within the virtual network, with address prefix `["10.1.1.0/24"]`.
- `resource "azurerm_subnet" "subnettwo"`: A second subnet named `subnettwo` within the virtual network, with address prefix `["10.1.2.0/24"]`.

The Explorer sidebar on the left shows the project structure for `TERRAFORMTEST`, including `.terraform`, `.terraform.lock.hcl`, `main.tf`, `terraform.tfstate`, `terraform.tfstate.backup`, `terraform.tfvars`, and `variable.tf`. The status bar at the bottom indicates the file is at line 22, column 2, using UTF-8 encoding and CRLF line endings.

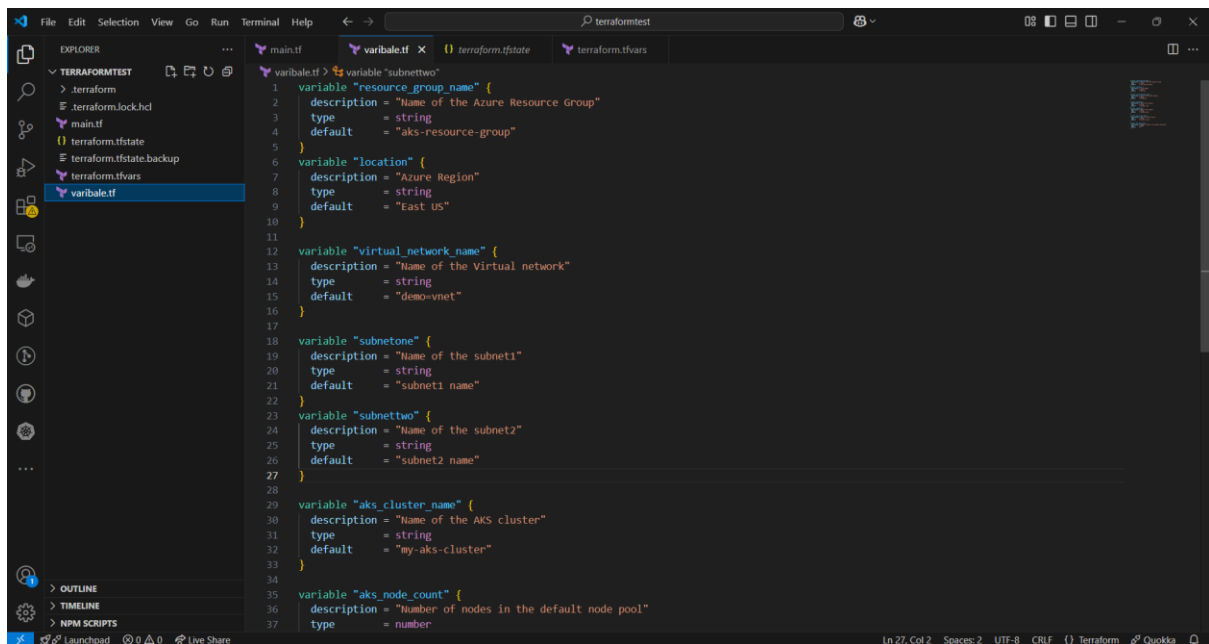


This screenshot shows the same Visual Studio Code editor with a different Terraform script for an Azure Kubernetes Cluster. The script defines the following resources:

- `resource "azurerm_kubernetes_cluster" "example"`: An AKS cluster named `example` in the location `azurerm_resource_group.example.location`, with DNS prefix `"aks-cluster"`.
- `default_node_pool`: Configuration for the default node pool with `name = "default"`, `node_count = 2`, and `vm_size = "Standard_DS2_v2"`.
- `identity`: Configuration for the cluster identity with `type = "SystemAssigned"`.
- `depends_on`: A dependency on `azurerm_subnet.subnet1` to ensure the subnet exists before the cluster is created.

The Explorer sidebar shows the same project structure as the previous screenshot. The status bar at the bottom indicates the file is at line 22, column 2, using UTF-8 encoding and CRLF line endings.

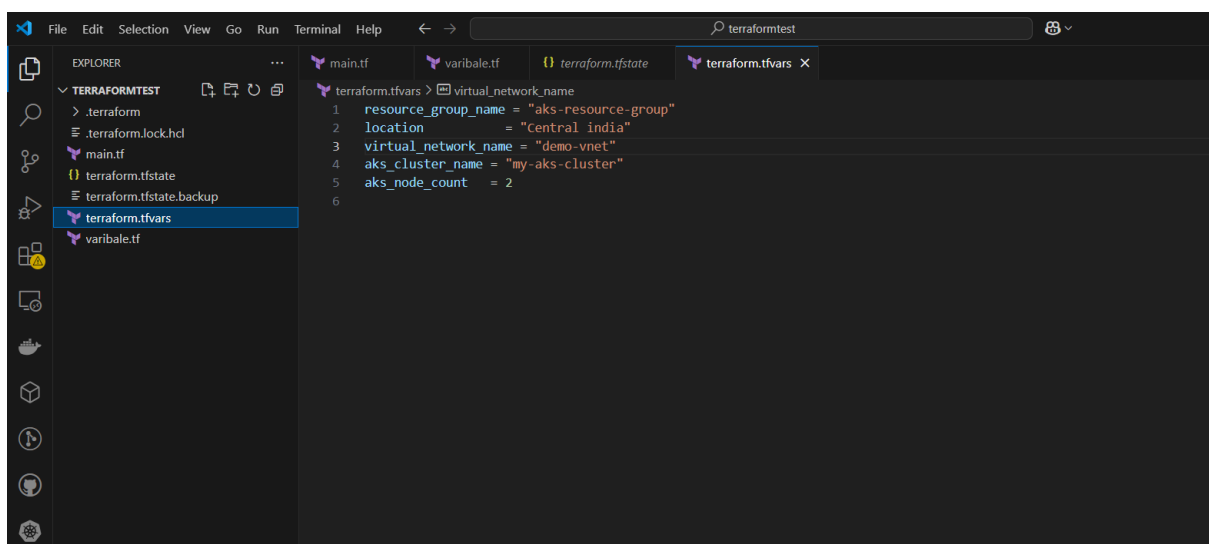
→ Create a Variable file based on the main code.



The screenshot shows the Visual Studio Code editor with the file explorer on the left displaying a project named 'TERRAFORMTEST'. The file explorer lists files: .terraform, .terraform.lock.hcl, main.tf, terraform.tfstate, terraform.tfstate.backup, terraform.tfvars, and variable.tf. The 'variable.tf' file is selected and open in the editor. The code in 'variable.tf' defines several Terraform variables:

```
1 variable "resource_group_name" {
2   description = "Name of the Azure Resource Group"
3   type       = string
4   default    = "aks-resource-group"
5 }
6 variable "location" {
7   description = "Azure Region"
8   type       = string
9   default    = "East US"
10 }
11
12 variable "virtual_network_name" {
13   description = "Name of the Virtual network"
14   type       = string
15   default    = "demo-vnet"
16 }
17
18 variable "subnetone" {
19   description = "Name of the subnet1"
20   type       = string
21   default    = "subnet1 name"
22 }
23 variable "subnettwo" {
24   description = "Name of the subnet2"
25   type       = string
26   default    = "subnet2 name"
27 }
28
29 variable "aks_cluster_name" {
30   description = "Name of the AKS cluster"
31   type       = string
32   default    = "my-aks-cluster"
33 }
34
35 variable "aks_node_count" {
36   description = "Number of nodes in the default node pool"
37   type       = number
38 }
```

→ Create a terraform var file to call the values

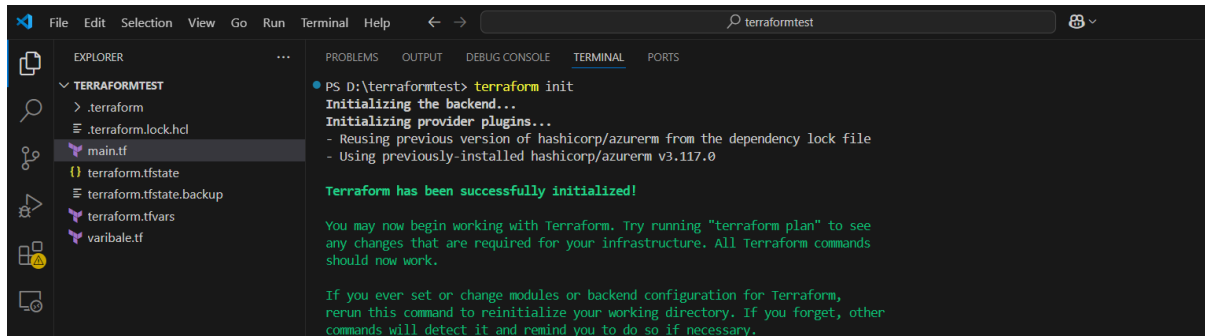


The screenshot shows the Visual Studio Code editor with the file explorer on the left. The file explorer lists files: .terraform, .terraform.lock.hcl, main.tf, terraform.tfstate, terraform.tfstate.backup, terraform.tfvars, and variable.tf. The 'terraform.tfvars' file is selected and open in the editor. The code in 'terraform.tfvars' assigns values to the variables defined in 'variable.tf':

```
1 resource_group_name = "aks-resource-group"
2 location            = "Central india"
3 virtual_network_name = "demo-vnet"
4 aks_cluster_name    = "my-aks-cluster"
5 aks_node_count      = 2
6
```


→ Before Initialize the terraform commands we need to connect to azure account with **az login** command.

The **terraform init** command is used to initialize a Terraform working directory.



```
File Edit Selection View Go Run Terminal Help
terraformtest

EXPLORER
TERRAFORMTEST
> .terraform
.terraform.lock.hcl
main.tf
terraform.tfstate
terraform.tfstate.backup
terraform.tfvars
variable.tf

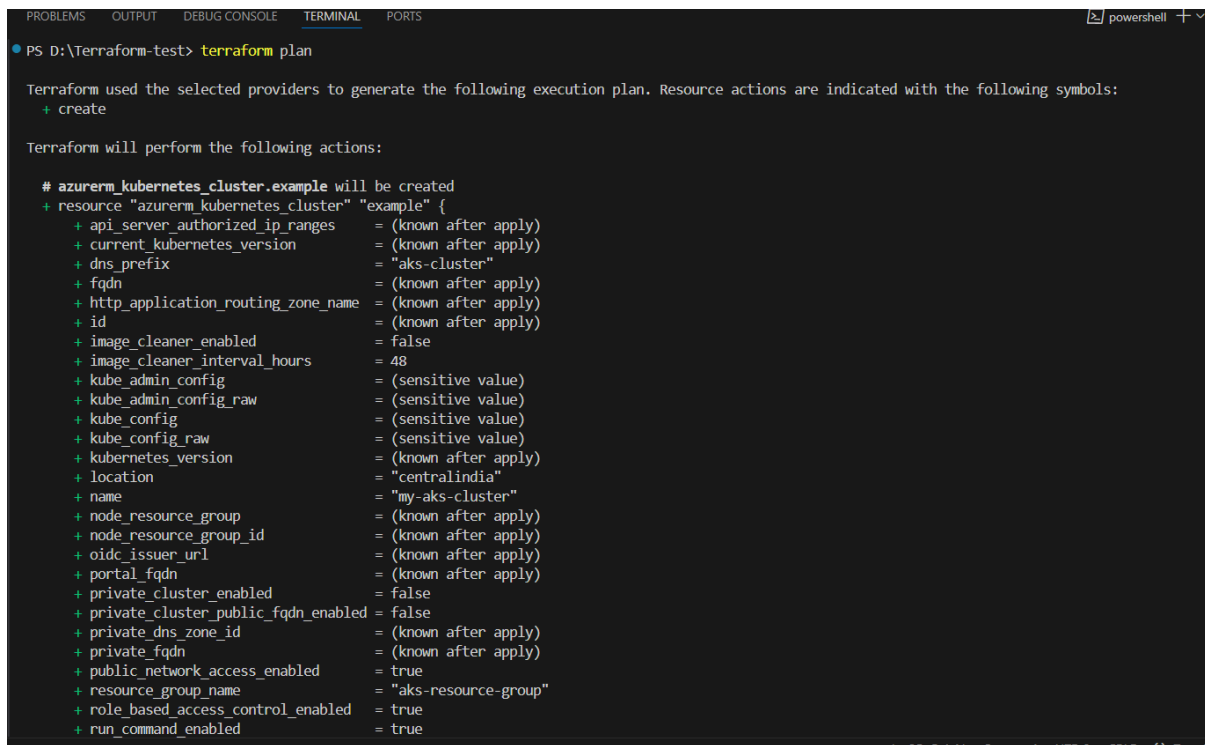
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS D:\terraformtest> terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Using previously-installed hashicorp/azurerm v3.117.0

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.
```

→ The **terraform plan** command is used to generate an execution plan.

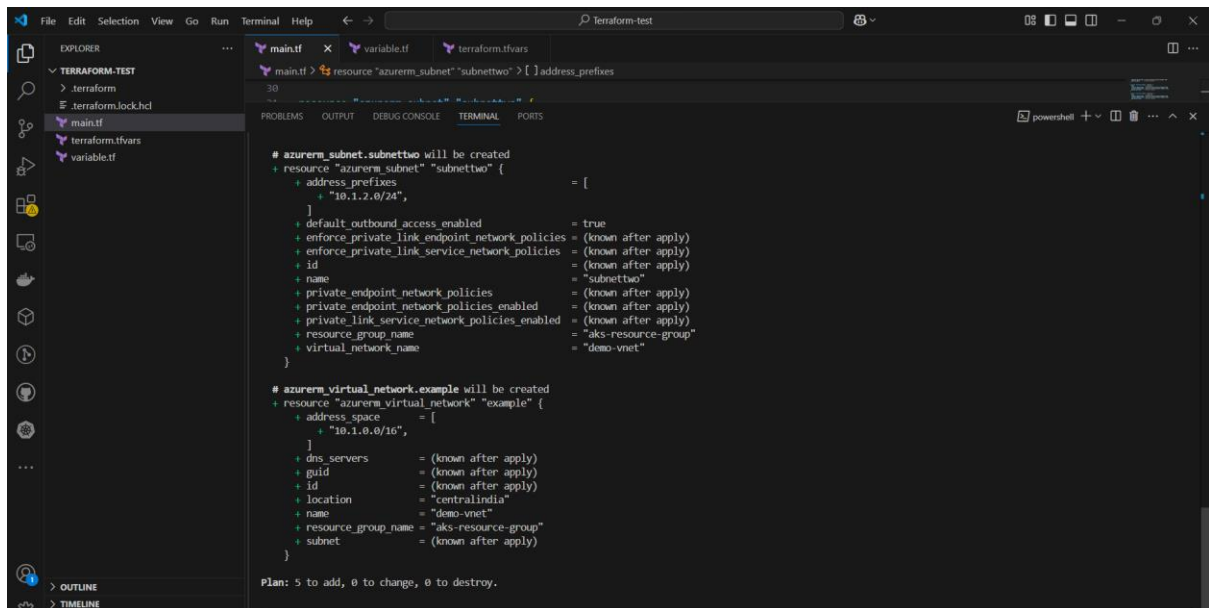


```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS powershell +
PS D:\Terraform-test> terraform plan

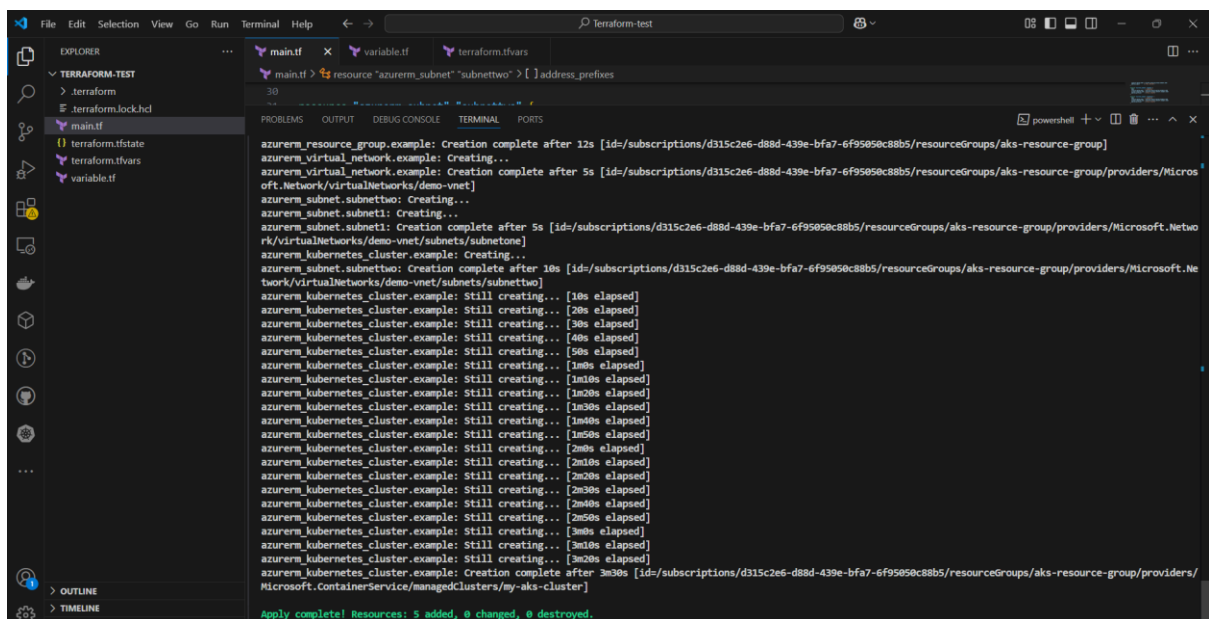
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
+ create

Terraform will perform the following actions:

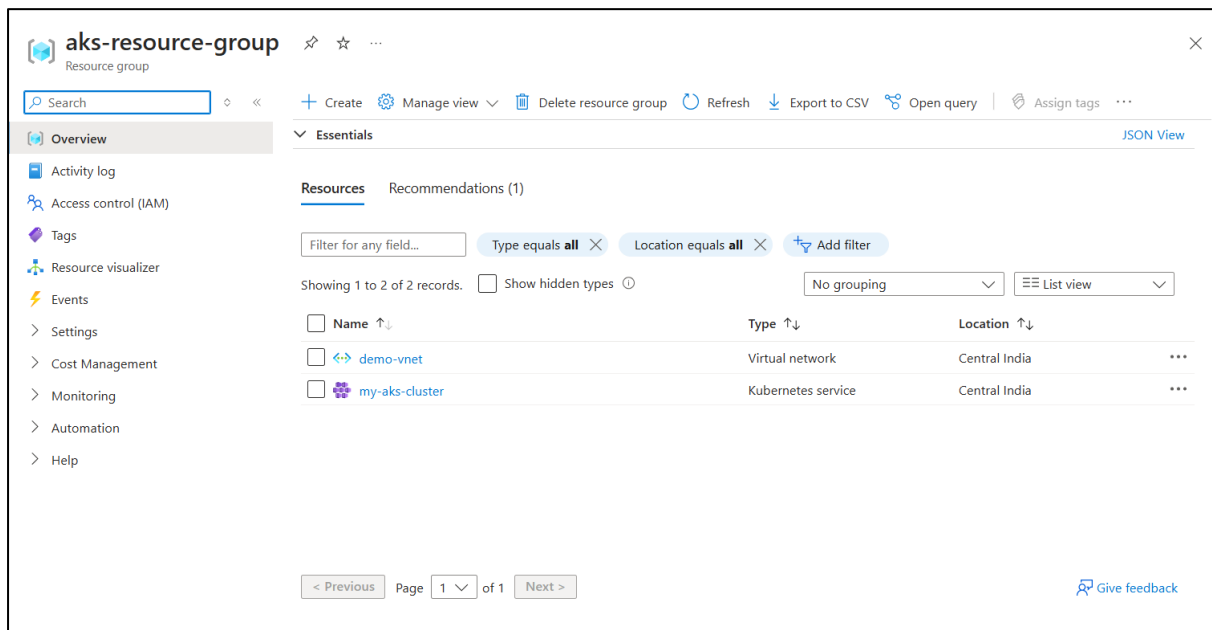
# azurerm_kubernetes_cluster.example will be created
+ resource "azurerm_kubernetes_cluster" "example" {
  + api_server_authorized_ip_ranges = (known after apply)
  + current_kubernetes_version      = (known after apply)
  + dns_prefix                      = "aks-cluster"
  + fqdn                            = (known after apply)
  + http_application_routing_zone_name = (known after apply)
  + id                              = (known after apply)
  + image_cleaner_enabled           = false
  + image_cleaner_interval_hours    = 48
  + kube_admin_config               = (sensitive value)
  + kube_admin_config_raw           = (sensitive value)
  + kube_config                    = (sensitive value)
  + kube_config_raw                 = (sensitive value)
  + kubernetes_version              = (known after apply)
  + location                        = "centralindia"
  + name                            = "my-aks-cluster"
  + node_resource_group              = (known after apply)
  + node_resource_group_id          = (known after apply)
  + oidc_issuer_url                 = (known after apply)
  + portal_fqdn                     = (known after apply)
  + private_cluster_enabled          = false
  + private_cluster_public_fqdn_enabled = false
  + private_dns_zone_id              = (known after apply)
  + private_fqdn                     = (known after apply)
  + public_network_access_enabled    = true
  + resource_group_name              = "aks-resource-group"
  + role_based_access_control_enabled = true
  + run_command_enabled              = true
```



→ The **terraform apply** command is used to apply changes to your infrastructure.



→ verify the Resources are Created.



6. Deployment to Azure Kubernetes Service:

az account set --subscription "<SUBSCRIPTION_ID>"

az aks get-credentials --resource-group <RESOURCE_GROUP> --name <CLUSTER_NAME>

→ Create a Deployment file to host the image to Azure Kubernetes service.

```
! Deploy.yaml 1 x ! service.yaml

! Deploy.yaml > {} spec > {} template > {} spec > [ ] containers > {} 0 > image
io.k8s.api.apps.v1.Deployment (v1@deployment.json)
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: lok-deploy
5    labels:
6      app: app-dev
7  spec:
8    replicas: 2
9    selector:
10     matchLabels:
11       app: app-dev
12   template:
13     metadata:
14       labels:
15         app: app-dev
16     spec:
17       containers:
18         - name: image
19           image: lokeshkoagiri/simpletimeserviceflask
20           imagePullPolicy: Always
21           ports:
22             - containerPort: 5000
23
24
```

→Run the Apply command to create a deployment to Cluster node.

```
PS C:\Users\ktssk\Desktop\aksapp> kubectl apply -f .\Deploy.yaml
deployment.apps/lok-deploy created
```

→Create a Service file to Expose the Public IP to Client with Load Balancer.

```
! service.yaml 1  ! service.yaml X
! service.yaml > {} spec > [ ] ports > {} 0 > # targetPort
io.k8s.api.core.v1.Service (v1@service.json)
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: lok-svc
5    labels:
6      app: app-dev
7      tier: Backend
8  spec:
9    type: LoadBalancer
10   selector:
11     app: app-dev
12   ports:
13     - name: http
14       port: 5000
15       targetPort: 5000
```

→Run the Apply command for service file to Expose the Public IP.

```
deployment.apps/lok-deploy created
PS C:\Users\ktssk\Desktop\aksapp> kubectl apply -f .\service.yaml
service/lok-svc created
```

→**kubectl get service** to view the IP address.

```
PS C:\Users\ktssk\Desktop\aksapp> kubectl get service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.0.0.1	<none>	443/TCP	24m
lok-svc	LoadBalancer	10.0.95.156	20.244.73.170	5000:32735/TCP	45s

→ Browse the Public IP address to view the webpage.

