**Challenge#1**

**A 3-tier environment is a common setup. Use a tool of your choosing/familiarity create these resources on a cloud environment (Azure/AWS/GCP). Please remember we will not be judged on the outcome but more focusing on the approach, style and reproducibility.**

**Sol:**

Yes 3-tier environment is a common setup for applications. In this 3-tier architecture there are three distinct layers.

1. Presentation layer: presentation layer as known as user interface (UI) layer, in which it will handle the user requests and shows the results.

Ex: Load Balancer

1. Application layer: Application layer contains the business logic. It will take the user requests from the presentation layer and interact with data layer if needed.

Ex: Virtual machine/Virtual machine Scale set.

1. Data layer: Data layer will store the application Data.

Ex: Azure SQL Database.

Comes to creating the resources I am familiar with Terraform (IaC) to create the Infrastructure in the Azure Cloud.

**I have Experience to writing Terraform code in Visual Studio Code, and I will use YAML to deploy the code to the Azure cloud.**

**# Declaring the Back end to store the tf state file in the Blob storage account.**

terraform {

backend "azurerm" {

resource\_group\_name = "resource\_group\_name"

storage\_account\_name = "storage\_account\_name"

container\_name = "tfstate"

key = "name\_of\_terraform\_tfstate\_file"

}

}

**# Define my cloud provider (azurerm)**

provider "azurerm" {

features {}

}

**# Creating the resource group**

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

name = "resource\_group\_name"

location = "East US" # Give location name as per your requirement

}

**# Create Azure Virtual Network**

resource "azurerm\_virtual\_network" "vnet1" {

name = "virtual\_machine\_name"

address\_space = ["10.0.0.0/16"]

location = "eastus" # Update with your desired region

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

}

**# Create Azure Subnet for the app tier**

resource "azurerm\_subnet" "app\_subnet" {

name = "app-subnet"

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

virtual\_network\_name = azurerm\_virtual\_network.vnet1.name

address\_prefixes = ["10.0.1.0/24"] # Update with your desired subnet CIDR

}

1. **Creating load balancer in Presentation layer**

**# Creating Azure Load Balancer for web tier (handling the user requests for application)**

resource "azurerm\_lb" "web\_lb" {

name = "load\_balancer\_for\_web"

location = "eastus" # Update with your desired region

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

frontend\_ip\_configuration {

name = "PublicIPAddress"

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

}

}

**# Create Azure Virtual Machines for web tier**

resource "azurerm\_virtual\_machine" "web\_instances" {

count = 2 # Update with the desired number of instances

name = "web-vm-${count.index}"

location = "eastus" # Update with your desired region

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

network\_interface\_ids = [azurerm\_network\_interface.web\_nic.id]

vm\_size = "Standard\_B1s” # Update with your desired VM size

storage\_image\_reference {

publisher = "Canonical"

offer = "UbuntuServer"

sku = "16.04-LTS"

version = "latest"

}

}

1. **Creating virtual machines in the Application Layer**

# Create Azure Virtual Machines for app tier

resource "azurerm\_virtual\_machine" "app\_instances" {

count = 2 # Update with the desired number of instances

name = "app-vm-${count.index}"

location = "eastus" # Update with your desired region

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

network\_interface\_ids = [azurerm\_network\_interface.app\_nic.id]

vm\_size = "Standard\_B2s" # Update with your desired VM size

storage\_image\_reference {

publisher = "Canonical"

offer = "UbuntuServer"

sku = "16.04-LTS"

version = "latest"

}

}

1. **Creating sql database in Data Layer to store the application data**

# Create Azure SQL Database for the data layer

resource "azurerm\_sql\_server" "database\_server" {

name = "database-server"

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

location = azurerm\_resource\_group.gg\_name.location

version = "12.0"

administrator\_login = "adminuser"

administrator\_login\_password = "password123"

}

resource "azurerm\_sql\_database" "database" {

name = "example-db"

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

location = azurerm\_resource\_group.rg\_name.location

server\_name = azurerm\_sql\_server.database\_server.name

edition = "Basic"

requested\_service\_objective\_name = "Basic"

collation = "SQL\_Latin1\_General\_CP1\_CI\_AS"

}

After creation of the resources, we need to establish the connections between the resources like below

1.Create Azure Virtual Network Rule to allow access from app tier

2. Connect the web tier to the application tier

3.Connect the application tier to the database tier

4. Configure database connection string in the application tier

**Next step is Writing the yaml code to deploy the resources in the azure cloud using Azure DevOps CICD.**

1. **Ccreate Azure DevOps Organization**
2. **Create the Project**
3. **Push the Code into Azure DevOps Repo**

name: $(Date:yyyymmdd)$(Rev:r) # Pipeline name

trigger:

branches:

include:

- main

pool:

vmImage: 'Window-latest'

steps:

- task: TerraformInstaller@0

displayName: "Install Terraform"

inputs:

terraformVersion: '1.3.9'

terraformDownloadLocation: 'https://releases.hashicorp.com/terraform'

- script: |

terraform init

terraform plan -out=terraform.tfplan

displayName: 'Terraform Plan'

- task: PublishBuildArtifacts@1

inputs:

pathtoPublish: '$(Build.SourcesDirectory)/terraform.tfplan'

artifactName: 'terraformPlan'

publishLocation: 'Container'

- task: DownloadPipelineArtifact@2

displayName: 'Download Terraform Plan'

inputs:

buildType: specific

buildVersionToDownload: 'latest'

project: 'Azure\_DevOps\_Project\_name' # replace with the name of your Azure DevOps Project

definition: 'Terraform Plan'

artifactName: 'terraformPlan'

path: '$(Pipeline.Workspace)'

- script: |

terraform apply $(Pipeline.Workspace)/terraform.tfplan

displayName: 'Terraform Apply'

**Note: We can separate the terraform plan and terraform apply into two different pipelines. By doing this, we can first check the plan to ensure it shows the desired output, and then we can run the Terraform apply pipeline."**

**Challenge#2**

We need to write code that will query the meta data of an instance within AWS or Azure or GCP and provide a json formatted output. The choice of language and implementation is up to you.

Bonus Points:

The code allows for a particular data key to be retrieved individually

Hints:

Aws Documentation (<https://docs.aws.amazon.com/>)

Azure Documentation (<https://docs.microsoft.com/en-us/azure/?product=featured>)

Google Documentation (<https://cloud.google.com/docs>)

**Sol:**

I am familiar with the Azure Cloud and writing the code in terraform. For example here I am choosing the azure web app (Azure App Service) Instance to to get the meta data and will write the terraform code to get the output in JSON format.

**Note: After creation of the resources in azure we can fetch the resources data using data block in the terraform.**

**Terraform code:**

provider "azurerm" {

features {}

}

data "azurerm\_app\_service" "web\_app\_metadata" {

name = "existing\_web\_app\_name"

resource\_group\_name = "WebApp\_resourcegroupname"

}

#Shows the web app metadata

output "web\_app\_metadata\_json" {

value = jsonencode(data.azurerm\_app\_service.web\_app\_metadata)

}

In this example, we are using the azurerm\_app\_service data source to fetch the metadata of the web app (Azure App Service) specified in the name and resource\_group\_name attributes. The output block then uses jsonencode to convert the fetched data into JSON format.

When you run Terraform, it will fetch the web app metadata and display the JSON-formatted output in the console.

You can also use the terraform apply command to create an output file containing the metadata in JSON format. Just redirect the output to a file like this:

terraform apply -auto-approve > web\_app\_metadata.json

This will create a file named web\_app\_metadata.json containing the JSON-formatted metadata of the specified Azure App Service (Web App).

To retrieve a specific data key individually, use the specific\_data\_key output block. Replace <desire\_data\_key> with the actual key you want to retrieve from the metadata.

**output "specific\_data\_key" {**

**value = data.azurerm\_app\_service.web\_app\_metadata.<desire\_data\_key>**

**}**

For example, if you want to retrieve the URL of the web app, the code looks like below

**output "specific\_data\_key" {**

**value = data.azurerm\_app\_service.web\_app\_metadata.default\_site\_hostname**

**}**

**Challenge#3**

**We have a nested object. We would like a function where you pass in the object and a key and get back the value.**

**The choice of language and implementation is up to you.**

**Example Inputs**

**object = {“a”:{“b”:{“c”:”d”}}}**

**key = a/b/c**

**object = {“x”:{“y”:{“z”:”a”}}}**

**key = x/y/z**

**value = a**

**Sol:**

Here I am select Terraform as a language.

**Terraform code:**

Provider “azurerm”{

Feature{}

}

locals {

object1 = {"a" = { "b" = { "c" = "d"} }

object2 = { "x" = {"y" = {"z" = "a"} }

}

function get\_value\_from\_nested\_object(object, key) {

var keys = split("/", key)

var current\_object = object

for key\_part in keys {

if current\_object[key\_part] == null {

return null

}

current\_object = current\_object[key\_part]

}

return current\_object

}

}

output "value\_example1" {

value = local.get\_value\_from\_nested\_object(local.object1, "a/b/c")

}

output "value\_example2" {

value = local.get\_value\_from\_nested\_object(local.object2, "x/y/z")

}