Lab Exercise 8– Terraform Multiple tfvars Files

Objective:

Learn how to use multiple tfvars files in Terraform for different environments.

Prerequisites:

- Terraform installed on your machine.
- Basic knowledge of Terraform configuration and variables.

Steps:

1. Create a Terraform Directory:

```
mkdir terraform-multiple-tfvars
cd terraform-multiple-tfvars
```

- Create Terraform Configuration Files:
- Create a file named main.tf:

 # main.tf

```
provider "aws" {
  region = var.region
}

resource "aws_instance" "example" {
  ami          = var.ami
  instance_type = var.instance_type
}
```

- Create a file named variables.tf:

 # variables.tf

```
variable "ami" {
   type = string
}

variable "instance_ty" {
   type = string
}
```

2. Create Multiple tfvars Files:

- Create a file named dev.tfvars:

 # dev.tfvars

```
dev.tfvars
my-ami = "ami-03235cc8fe4d9bf1e"
my-instance-type = "t3.micro"
```

- Create a file named prod.tfvars:

 # qa.tfvars

```
qa.tfvars
my-ami = "ami-03235cc8fe4d9bf1e"
my-instance-type = "t3.large"
```

In these files, provide values for the variables based on the environments.

3. Initialize and Apply for Dev Environment:

- Run the following Terraform commands to initialize and apply the configuration for the dev environment:

terraform init

terraform apply -var-file=dev.tfvars

```
iamyo@Acernitro MINGW64 /d/terraform-demo
$ terraform apply -var-file=dev.tfvars

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:

  # aws_instance.my_instance will be created
  + resource "aws_instance" "my_instance" {
      + ami                                  = "ami-03235cc8fe4d9bf1e"
      + arn                                  = (known after apply)
      + associate_public_ip_address          = (known after apply)
      + availability_zone                    = (known after apply)
      + cpu_core_count                       = (known after apply)
      + cpu_threads_per_core                 = (known after apply)
      + disable_api_stop                     = (known after apply)
      + disable_api_termination              = (known after apply)
      + ebs_optimized                        = (known after apply)
      + get_password_data                    = false
      + host_id                              = (known after apply)
      + host_resource_group_arn              = (known after apply)
      + iam_instance_profile                 = (known after apply)
      + id                                   = (known after apply)
```

4. Initialize and Apply for Prod Environment:

- Run the following Terraform commands to initialize and apply the configuration for the prod environment:

terraform init

terraform apply -var-file=qa.tfvars

```
iamyo@Acernitro MINGW64 /d/terraform-demo
$ terraform apply -var-file=qa.tfvars
aws_s3_bucket.my_bucket: Refreshing state... [id=my-demo-s3-bucket123]
aws_instance.my_instance: Refreshing state... [id=i-04c52ddf8cb8f8a2b]

Terraform used the selected providers to generate the following execution plan.
Resource actions are indicated with the following symbols:
  ~ update in-place

Terraform will perform the following actions:

  # aws_instance.my_instance will be updated in-place
  ~ resource "aws_instance" "my_instance" {
        id                                   = "i-04c52ddf8cb8f8a2b"
      ~ instance_type                        = "t3.micro" -> "t3.large"
        tags                                 = {
            "Name" = "UPES-EC2-Instance"
        }
        # (38 unchanged attributes hidden)

        # (8 unchanged blocks hidden)
    }

Plan: 0 to add, 1 to change, 0 to destroy.
```

5. Test and Verify:

- Observe how different tfvars files are used to set variable values for different environments during the apply process.
- Access the AWS Management Console or use the AWS CLI to verify the creation of resources in the specified regions and instance types.

6. Clean Up:

- After testing, you can clean up resources:

```
terraform destroy -var-file=dev.tfvars
```

```
        }

      - versioning {
          - enabled    = false -> null
          - mfa_delete = false -> null
        }
    }

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

Do you really want to destroy all resources?
  Terraform will destroy all your managed infrastructure, as shown above.
  There is no undo. Only 'yes' will be accepted to confirm.

  Enter a value: yes

aws_s3_bucket.my_bucket: Destroying... [id=my-demo-s3-bucket123]
aws_instance.my_instance: Destroying... [id=i-04c52ddf8cb8f8a2b]
aws_s3_bucket.my_bucket: Destruction complete after 1s
aws_instance.my_instance: Still destroying... [id=i-04c52ddf8cb8f8a2b, 10s elapsed]
aws_instance.my_instance: Destruction complete after 14s

Destroy complete! Resources: 2 destroyed.

iamyo@Acernitro MINGW64 /d/terraform-demo
$
```

terraform destroy -var-file=qa.tfvars

```
iamyo@Acernitro MINGW64 /d/terraform-demo
$ terraform destroy -var-file=qa.tfvars

No changes. No objects need to be destroyed.

Either you have not created any objects yet or the existing objects were already
deleted outside of Terraform.

Destroy complete! Resources: 0 destroyed.

iamyo@Acernitro MINGW64 /d/terraform-demo
$
```

- Confirm the destruction by typing yes.

7. Conclusion:

This lab exercise demonstrates how to use multiple tfvars files in Terraform to manage variable values for different environments. It allows you to maintain separate configuration files for different environments, making it easier to manage and maintain your infrastructure code. Experiment with different values in the dev.tfvars

and prod.tfvars files to observe how they impact the infrastructure provisioning process for each environment.