Terraform Notes

Introduction to Terraform

Terraform is an open-source Infrastructure as Code (IaC) tool created by HashiCorp. It allows you to define, provision, and manage infrastructure across various cloud providers using a declarative configuration language (HCL - HashiCorp Configuration Language).

Key Benefits:

Declarative Language: You describe what infrastructure you need, and Terraform builds it for you.
Multi-Cloud Compatibility: Supports providers like AWS, Azure, Google Cloud, and others.
State Management: Keeps track of resources to handle infrastructure changes.
Execution Plans: Provides a preview of changes before applying them.

Terraform Core Concepts

1. Providers

Providers are plugins used to interact with APIs of cloud platforms like AWS, Azure, or GCP. Each provider has its own set of resources and data sources.

```
provider "aws" {
  region = "us-east-1"
}
```

2. Resources

Resources are the main components in your infrastructure, like virtual machines, storage, databases, etc.

```
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = "t2.micro"
}
```

3. Variables

Variables are placeholders for values to make configurations reusable and parameterized.

```
variable "instance_type" {
  default = "t2.micro"
}
```

4. Outputs

Outputs are used to display information after a configuration is applied, such as IP addresses or URLs.

```
output "instance_ip" {
  value = aws_instance.example.public_ip
}
```

5. State

Terraform uses a state file to keep track of infrastructure resources, enabling incremental updates.

Basic Terraform Workflow:

Write Configuration: Create .tf files containing your desired infrastructure in HCL syntax.
Initialize: Run terraform init to initialize the project and download provider plugins.
Plan: Run terraform plan to see what changes Terraform will make to your infrastructure.
Apply: Run terraform apply to execute the plan and provision resources.
Destroy: Run terraform destroy to tear down all infrastructure defined in the configuration.

Terraform Commands

1. terraform init

Initializes the working directory with plugins, modules, and backend setup.

terraform init

2. terraform plan

Shows a preview of what actions Terraform will take when apply is run.

terraform plan

3. terraform apply

Applies the changes required to reach the desired state of the configuration.

terraform apply

4. terraform destroy

Destroys all resources managed by Terraform in the current configuration.

terraform destroy

5. terraform fmt

Formats configuration files for readability and best practices.

terraform fmt

6. terraform validate

Validates the syntax and configuration of the files without deploying anything.

terraform validate

Writing a Basic Terraform Configuration

Define the Provider – Specify which cloud provider you'll be using.
Create Resources – Define the infrastructure components you want.
Use Variables – Parameterize the configuration for flexibility.
Output Values – Define output values to get useful information.

Example: Deploying an EC2 Instance on AWS

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = “t2.micro”
  tags = {
    Name = "ExampleInstance"
  }
}
:wq
```

03496740587

```
> terraform init
> terraform plan
> terraform apply
```

If you want to delete the resources:

```
> terraform state list
> terraform destroy -target=”target-id”
```

Defining Variables

Variables can be specified in multiple ways:

```
> vim main.tf
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = var.instance_type
  tags = {
    Name = "ExampleInstance"
  }
}

variable "instance_type" {
  default = "t2.micro"
   type = string
  description = "Type of EC2 instance to deploy"
}

:wq
> terraform apply
```

Terraform Var Files:

```
> vim main.tf
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
count = var.instance_count
```

```
  ami           = "ami-123456"
  instance_type = var.instance_type
  tags = {
    Name = "ExampleInstance"
  }
}
:wq
> vim variable.tf
variable “instance_count” {
description = “*”
type= number
default=3

variable “instance_type” {
description = “*”
type= string
default=”t2.micro”
:wq
> terraform init
> terraform plan
> terraform apply –auto-approve
```

Second Method:

```
vim main.tf
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
count = var.instance_count
  ami           = "ami-123456"
  instance_type = var.instance_type
  tags = {
    Name = "ExampleInstance"
  }
}
:wq
Vim variable.tf
variable “instance_count” {
}
```

03496740587

```
variable “instance_type” {
}
:wq


vim dev.tfvars
instance_count = 1
instance_type = “t2.micro”
:wq

> vim test.tfvars
instance_count = 2
instance_type = “t2.medium”

:wq

> terraform apply –auto-approve -var-file=”dev.tfvars”
> terraform apply –auto-approve -var-file=”test.tfvars”
> terraform destroy –auto-approve -var-file=”dev.tfvars”
```

Terraform CLI:

```
> vim main.tf

# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = var.instance_type
  tags = {
    Name = "ExampleInstance"
  }
}

variable “instance_type” {
}

:wq

> terraform apply –auto-approve
```

03496740587

Enter Value: t2.micro

OR

terraform apply –auto-approve -var=”instance_type=t2.micro”

Terraform output:

It is used to print information of resource instance.

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = “t2.micro”
  tags = {
    Name = "ExampleInstance"
  }
}

output  “dmh” {
value = [aws_instance.dmh.public_ip,
aws_instance.dmh.private_ip,aws_instance.dmh.public_dns,aws_instance.dmh.private_dns]
}

:wq
```

Terraform Import:

It is used to import and track the resources which are created manually.

> First create an instance manually.
> copy instance id.
> vim main.tf

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
}
:wq
```

> terraform import aws_instance.example <past instance id here>

03496740587

Terraform s3 bucket:

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_s3_bucket" "example" {
Bucket = “anyuniquebucketname”
}
:wq

> terraform apply
```

Terraform ebs volume:

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_ebs_volume" "example" {
Size = 20
availability_zone = “us-east-1a”
}
:wq
> terraform apply
```

Terraform iam user:

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_iam_user" "example" {
name = “dmh”
}
:wq
> terraform apply
```

Terraform Lifecycle:

03496740587

It is used to keep our resources secure without destroying them.

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = “t2.micro”
  tags = {
    Name = "ExampleInstance"
  }
lifecycle {
Prevent_destroy = true
}
}
:wq
> terraform apply
```

Terraform commit:

If we put commot , it will not work for that action.

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = “t2.micro”
  tags = {
    Name = "ExampleInstance"
  }
}

/*resource "aws_instance" "example1" {
  ami           = "ami-123456"
  instance_type = “t2.medium”
  tags = {
    Name = "ExampleInstance1"
  }
}*/
:wq
> terraform apply
```

03496740587

Terraform FMT:

It is used to provide indentation for terraform.

```
> terraform fmt
```

Terraform Local Resources:

It is used to create local resources with the help of terraform file.

```
> vim main.tf
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "local_file" "example" {
  filename        = "abc.txt"
  content= “hello world!”
   }
:wq
> terraform init
> terraform apply
```

Terraform Workspaces:

What are Workspaces?

Workspaces are a way to maintain multiple, isolated state files for a single Terraform configuration.

They allow you to manage different environments (like dev, staging, and production) within a single configuration setup.

By default, every Terraform configuration has a single workspace named default.

Why Use Workspaces?

Workspaces are useful for managing different environments without creating separate directories or configurations.
They keep the state for each environment separate, which is helpful when deploying similar infrastructure with slight differences (like instance sizes or numbers).

Common Commands for Workspaces

Create a New Workspace

```
terraform workspace new <workspace_name>
```

Example:

```
terraform workspace new dev
```

Switch Between Workspaces

```
terraform workspace select <workspace_name>
```

Example:

```
terraform workspace select prod
```

List All Workspaces

```
terraform workspace list
```

Show the Current Workspace

```
terraform workspace show
```

Delete a Workspace

You can delete a workspace, but only if it is not in use.

```
terraform workspace delete <workspace_name>
```

Terraform Taint

What is Terraform Taint?

Taint is a command that marks a specific resource for recreation.
By marking a resource as tainted, you tell Terraform to destroy and recreate that resource during the next apply operation.
Useful for cases where a resource is malfunctioning or you want to force an update without changing the configuration file.

Why Use Terraform Taint?

When you have a resource that’s problematic, such as a misconfigured or corrupted resource, and you want to replace it without making configuration changes.
Useful for testing to see if recreating a resource would solve issues.

Commands for Taint and Untaint

Mark a Resource as Tainted

```
terraform taint <resource_type.resource_name>
```

Example:

```
terraform taint aws_instance.example
```

Remove the Taint on a Resource

If you want to remove the taint marking before applying, you can use untaint.

```
terraform untaint <resource_type.resource_name>
```

Example:

```
terraform untaint aws_instance.example
```

Apply Changes to Recreate the Tainted Resource

After marking a resource as tainted, run:

terraform apply

Terraform alias and providers:

It is used to create different resources in different regions with the help of same file.

```
# 1. Define Provider
provider "aws" {
  region = "us-east-1"
}

resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = “t2.micro”
  tags = {
    Name = "ExampleInstance"
  }
}

# 1. Define Provider
provider "aws" {
  region = "ap-southeast-1"
alias = “tokyo”
}

resource "aws_instance" "example" {
 Provider = aws.tokyo
  instance_type = “t2.medium”
  tags = {
    Name = "ExampleInstance1"
  }
}
:wq
> terraform apply
```

03496740587

Example configuration for S3 remote state:

Google: terraform s3 backened

Backend type: s3 | terraform

Copy the “”example configuration”

Create new bucket on aws and uses the same name of the bucket in the terraform file.

```
> vim main.tf
provider “aws” {
region = “us-east-1”
}

terraform {
  backend "s3" {
    bucket = "my-terraform-state"
    key    = "prod/terraform.tfstate"
    region = "us-east-1"
  }
}
resource "aws_instance" "example" {
  ami           = "ami-123456"
  instance_type = “t2.micro”
  tags = {
    Name = "ExampleInstance"
  }
}
:wq
> terraform apply
```

Note: If you delete the state file, you can get it from s3 bucket.

Terraform Dynamics:

It is used to reduce the length of code and used for reusability of code in loop.

```
> vim main.tf
provider “aws” {
}

Locals {
Ingress_rules = [{
Port = 443
description = “ingress rul for port 443”
```

03496740587

```
},
{
Port 8080
Description = “ingrss rule for port 8080”
}]
}
```