Terraform for Infrastructure Management

Terraform is known as "Infrastructure management" because it is an open-source tool that allows users to define and provision infrastructure resources in a declarative and automated manner. With Terraform, you can write code to describe the desired state of your infrastructure, such as virtual machines, networks, storage, and more, using a domain-specific language called HashiCorp Configuration Language (HCL) or JSON.

Once you've defined your infrastructure as code, terraform takes care of managing the entire lifecycle of these resources. It can create, modify, and delete infrastructure elements to match the desired state defined in the code. This means you can easily spin up and manage complex infrastructure setups, ensuring consistency across different environments, and easily make changes in a version-controlled and collaborative manner.

As a developer, we employ a structured approach to manage our infrastructure configuration efficiently. This involves organizing our codebase into multiple manifest files, each serving a specific purpose:

Provider Configuration (provider.tf):

In this file, we define the cloud provider we are using to deploy our infrastructure. We ensure that it contains the necessary configuration settings to establish a connection with the chosen cloud platform. Here, we specify the region, access keys, or other relevant details required to interact with the cloud services.

```
# Downloading Driver For AWS & Terraform Communication

terraform {
  required_version = "~> 1.2"
  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.0"
    }
  }
}
```

Variable Definitions (variables.tf):

In the variables.tf file, we define all the variables used in our infrastructure configuration. This enables us to make our code more flexible and adaptable by easily changing values without modifying the main code.

Like Here I have mentioned Security Group, Default Region, Amazon Machine Image ID, Default Key attached in the OS, Instance Type based on the workspace, Instance State which can be controlled for running or stopping the instance.

```
# Security Group
variable "mySecurityGroup" {}

# Default Region
variable "defaultRegion" {}

# AMI ID
variable "amiId" {}

# Default Key
variable "defaultKey" {}

# Default Instance Type
variable "defaultInstanceType" {
    type = map
    default = {
        dev = "t2.micro"
        test ="t2.small"
        prod = "t2.large"
    }
}

# Default Instance State
variable "defaultInstanceState" {}
```

In the provided Terraform code, the variable "defaultInstanceType" defines a variable called "defaultInstanceType" of type map. Let's elaborate on what the map type is and how it is used in this context:

Type:

In Terraform, variables can have various types to represent different kinds of data. The type attribute in the variable definition specifies the data type of the variable's value. In this case, the type is set to "map."

Map Type:

A map is a collection type in Terraform that allows you to associate keys with values. It is similar to dictionaries or associative arrays in other programming languages. In the context of variables, a map allows you to define a set of key-value pairs.

Default Value:

The default attribute in the variable definition specifies the default value for the variable if no value is explicitly provided when running Terraform. In this case, the default value is a map containing three key-value pairs:

"dev": The key "dev" is associated with the value "t2.micro".

"test": The key "test" is associated with the value "t2.small".

"prod": The key "prod" is associated with the value "t2.large".

These key-value pairs define the default instance types to be used for different environments: "dev," "test," and "prod."

The variable "defaultInstanceType" allows you to set instance types dynamically based on the selected environment. When using this variable, you can provide a value for the variable in your Terraform configuration or let it use the default map defined in the variable declaration.

In Resource_EC2_EBS.tf file

```
# Launching AWS Instance
resource "aws_instance" "web" {
    ami           = var.amiId
    key_name      = var.defaultKey
    # instance_type = var.defaultInstanceType

    # Retrive the Instance Type from Dev Variale
    # instance_type = var.defaultInstanceType["dev"]

    # Retrive the Instance Type from Default Workspace
    instance_type = lookup(var.defaultInstanceType, terraform.workspace)
    vpc_security_group_ids  = [var.mySecurityGroup]
    count        = 2
    tags = {
      Name = "OS By TF- ${count.index + 1}"
    }
}
```

```
# Creating 1 Storage Device of 1 GB in Size in Region ap-south-1 with name
Webserver Volume
resource "aws_ebs_volume" "MyVolume"{
  availability_zone = aws_instance.web.availability_zone
  size              = 1
  tags = {
    Name = "WebServer Volume"
  }
# If AWS EC2 is created then Create new Harddisk
  depends_on = [ aws_instance.web ]
}

# Attaching the Harddisk to the OS
```

```
resource "aws_volume_attachment" "ebs_att" {
  device_name = "/dev/sdh"
  volume_id   = aws_ebs_volume.MyVolume.id
  instance_id = aws_instance.web.id
  force_detach = true
# Meta arguments & meta resources
# Depend on when EBS Volume is attached
  depends_on = [aws_ebs_volume.MyVolume]
}
```

Launching AWS Instance:

The first resource block defines an AWS EC2 instance. It creates a virtual machine in the Amazon Elastic Compute Cloud (EC2) service.

ami: The ID of the Amazon Machine Image (AMI) used to create the instance. It is specified via the variable var.amiId.

key_name: The name of the AWS key pair to associate with the instance. The value is taken from the variable var.defaultKey.

instance_type: The instance type, which determines the compute capacity of the instance. It is dynamically retrieved based on the Terraform workspace using the lookup function with var.defaultInstanceType as the source.

vpc_security_group_ids: A list of security group IDs to associate with the instance, taken from the variable var.mySecurityGroup.

tags: A set of key-value pairs used to tag the instance for better organization and identification. Here, it has a single tag "Name" with the value "OS By TF".

Creating an EBS Volume:

The second resource block creates an Amazon Elastic Block Store (EBS) volume, which provides persistent block-level storage for the EC2 instance.

availability_zone: The availability zone in which the EBS volume should be created. It is set to the same availability zone as the EC2 instance (aws_instance.web.availability_zone).

size: The size of the EBS volume in gigabytes, set to 1 GB in this case.

tags: Similar to the instance, the EBS volume is also tagged with a "Name" tag set to "WebServer Volume".

Dependency on EC2 Instance:

The depends_on attribute is used in both resource blocks. It specifies that the EBS volume resource depends on the successful creation of the EC2 instance resource. This ensures that Terraform creates the EBS volume only after the EC2 instance is successfully created.

Attaching the EBS Volume to the EC2 Instance:

The third resource block defines the attachment of the EBS volume to the EC2 instance.

device_name: The device name to attach the volume to, set to "/dev/sdh".

volume_id: The ID of the EBS volume to attach, taken from aws_ebs_volume.MyVolume.id.

instance_id: The ID of the EC2 instance to which the EBS volume should be attached, taken from aws_instance.web.id.

force_detach: A boolean value that, if set to true, forces the detachment of the volume from any previous attached instance before attaching it to the current instance.

Instance_state.tf

```
# Current Status of the Instance
resource "aws_ec2_instance_state" "Current_state" {
  instance_id = aws_instance.web.id
  state       = var.defaultInstanceState
}
```

The provided Terraform code defines an AWS EC2 instance state data source, which allows you to retrieve information about the current state of an EC2 instance that has already been created outside of the current Terraform configuration. It does not create or modify any resources; instead, it retrieves the current state of an existing EC2 instance for informational purposes.

Defining the EC2 Instance State Data Source:

The resource block starts with resource and declares an aws_ec2_instance_state data source. This type of resource allows you to fetch information about an existing EC2 instance.

Instance ID:

The attribute instance_id is set to the ID of the EC2 instance for which you want to retrieve the current state. It uses the ID of the EC2 instance created earlier in the Terraform code, which is represented by aws_instance.web.id.

State Variable:

The attribute state is set to a variable named var.defaultInstanceState. This variable likely holds a specific EC2 instance state value, such as "running," "stopped,". The purpose of using a variable here is to allow flexibility in specifying the desired instance state when querying the data source.

For example, if you have a variable defaultInstanceState set to "running," the data source will provide information about whether the EC2 instance with the specified ID is currently running or not.

Null_Resource-webserverSetup.tf

```
resource "null_resource" "webpage-Configure" {
    # Force Replacement - Every Time
    # We can use timestamp(). Everytime we run the manifest file it will run
on new time.
    triggers = {
        CurrentTime = timestamp()
    }
    # Login Inside the OS by ssh using Username and Private Key
    connection {
          user        = "ec2-user"
          type        = "ssh"
          private_key = file("C:/Users/Himanshu Kumar/Documents/Terraform/Day
1/HimanshuTF.pem")
          host    = aws_instance.web.public_ip
    }

    # Downloading the Software Inside the OS
    provisioner "remote-exec" {
        inline = [
            "sudo yum -y install httpd",
            "sudo systemctl enable httpd --now"
            ]
    }
}
```

The provided Terraform code defines a `null_resource` resource block that represents a placeholder resource in Terraform. It is typically used for executing actions or running provisioners that do not directly relate to a specific infrastructure resource. Let's break down the code and explain each part:

1. Defining the `null_resource` Resource:

The code starts by declaring a `null_resource` named "webpage-Configure". This resource acts as a container for arbitrary actions or provisioners that need to be executed as part of the Terraform configuration.

2. Triggers for Forcing Replacement:

The `triggers` block allows you to set dependencies or force actions to happen when specific changes occur. In this case, the trigger sets the `CurrentTime` attribute to the current timestamp using the `timestamp()` function. This is done to force the `null_resource` to be recreated every time the Terraform configuration is applied. By doing so, it ensures that the actions defined in the provisioner are executed on every run, regardless of whether any other resources have changed.

3. SSH Connection Configuration:

The `connection` block defines the SSH connection details used for logging inside the EC2 instance provisioned earlier. It sets up the connection parameters for running remote-exec provisioners.

- `user`: The SSH username to use for the connection. In this case, it is set to "ec2-user."

- `type`: The type of connection, which is set to "ssh" in this case.

- `private_key`: The path to the private key file used for SSH authentication. It points to the "HimanshuTF.pem" file located on the local system.

- `host`: The public IP address of the EC2 instance, fetched from the `aws_instance.web.public_ip` attribute.

4. Remote Execution Provisioner:

The `provisioner "remote-exec"` block defines a provisioner that will run remote commands on the EC2 instance using SSH.

- `inline`: It contains a list of commands to be executed on the remote EC2 instance. In this case, two commands are specified: one to install the "httpd" software package using `yum`, and another to enable and start the "httpd" service using `systemctl`.

Overall, this Terraform code sets up a `null_resource` that runs SSH commands on an EC2 instance to install and start the "httpd" web server software. The resource is forced to be recreated on every Terraform run due to the trigger set with the current timestamp, ensuring the provisioner always runs.

Null_resource-Inventory.tf

```
resource "null_resource" "Ansible_Inventory" {
    triggers = {
        CurrentTime = timestamp()
    }
    # Printing the IP Address in the Inventory
    provisioner "local-exec" {
        command =  "echo ${aws_instance.web.public_ip} > inventory"
    }
    # If AWS EC2 Launch then Run this File
    depends_on = [aws_instance.web]
}
```

The provided Terraform code defines a null_resource resource block named "Ansible_Inventory." Similar to the previous examples, this resource acts as a placeholder to perform actions or execute provisioners that are not directly related to infrastructure resources. Let's go through the code and explain each part:

Defining the null_resource Resource:

The code starts by declaring a null_resource named "Ansible_Inventory."

Triggers for Forcing Replacement:

The triggers block sets up a trigger with the CurrentTime attribute, which is set to the current timestamp using the timestamp() function. As in the previous example, this is used to force the null_resource to be recreated every time the Terraform configuration is applied. It ensures that the provisioner inside this resource is executed on every Terraform run.

Local Execution Provisioner:

The provisioner "local-exec" block defines a provisioner that will run a local command on the machine where Terraform is executed.

command: It specifies the command to be executed. In this case, the command is to echo (print) the public IP address of the EC2 instance (aws_instance.web.public_ip) and redirect the output to a file named "inventory."

Dependency on AWS EC2 Instance:

The depends_on attribute ensures that this null_resource is created only after the aws_instance.web EC2 instance has been successfully created.

Overall, the purpose of this Terraform configuration is to create a null_resource that runs a local command to store the public IP address of the EC2 instance in an "inventory" file. This is a common practice when working with Ansible, where the "inventory" file specifies the hosts on which Ansible should execute tasks. By using Terraform's null_resource and local-exec provisioner, you can integrate the process of generating an Ansible inventory with the Terraform configuration.

Keep in mind that this example is just a simple illustration of how you can use Terraform to generate an Ansible inventory file. Depending on your use case, you may need to modify the provisioner command or perform additional configuration tasks to customize the inventory file further.

Null_resource-HardDiskFormat.tf

```
resource "null_resource" "Harddisk_Format" {
    # Login Inside the OS by ssh using Username and Private Key
    connection {
          user        = "ec2-user"
          type        = "ssh"
          private_key = file("C:/Users/Himanshu Kumar/Documents/Terraform/Day
1/HimanshuTF.pem")
          host    = aws_instance.web.public_ip
    }
    # Downloading the Software Inside the OS
    provisioner "remote-exec" {
        inline = [
            "sleep 10",
            "sudo mkfs.ext4 /dev/xvdh",
            "sudo mount /dev/xvdh /var/www/html",
            "sudo sh -c 'echo Hello Their > /var/www/html/index.html'"

            ]
    }
    depends_on = [null_resource.webpage-Configure]
}
```

The provided Terraform code defines a `null_resource` resource block named "Harddisk_Format." Let's go through the code and explain each part:

1. Defining the `null_resource` Resource:

The code starts by declaring a `null_resource` named "Harddisk_Format."

2. SSH Connection Configuration:

The `connection` block defines the SSH connection details used for logging inside the EC2 instance provisioned earlier. It sets up the connection parameters for running remote-exec provisioners.

- `user`: The SSH username to use for the connection. In this case, it is set to "ec2-user."

- `type`: The type of connection, which is set to "ssh" in this case.

- `private_key`: The path to the private key file used for SSH authentication. It points to the "HimanshuTF.pem" file located on the local system.

- `host`: The public IP address of the EC2 instance, fetched from the `aws_instance.web.public_ip` attribute.

3. Remote Execution Provisioner:

The `provisioner "remote-exec"` block defines a provisioner that will run remote commands on the EC2 instance using SSH.

- **`inline`:** It contains a list of commands to be executed on the remote EC2 instance. In this case, four commands are specified:

- **"sleep 10":** It adds a delay of 10 seconds before executing the next commands. This may be useful to ensure that the EC2 instance is fully ready to execute the following commands.

- **"sudo mkfs.ext4 /dev/xvdh":** It formats the EBS volume attached to the EC2 instance as an ext4 filesystem. The `/dev/xvdh` represents the device name of the attached volume.

- **"sudo mount /dev/xvdh /var/www/html": It** mounts the newly formatted EBS volume to the `/var/www/html` directory on the EC2 instance. This could be useful if you want to use the EBS volume for web server data, for example.

- **"sudo sh -c 'echo Hello Their > /var/www/html/index.html'":** It creates an index.html file with the content "Hello Their" in the `/var/www/html` directory.

4. Dependency on Another `null_resource`:

The `depends_on` attribute ensures that this `null_resource` is created only after another `null_resource` named "webpage-Configure" is successfully created. The presence of this dependency suggests that the "Harddisk_Format" provisioner should run after the "webpage Configure" provisioner has completed its execution.

Overall, this Terraform code sets up a `null_resource` that runs SSH commands on an EC2 instance to format a mounted EBS volume and populate it with some sample data.

Null_resource-GetContentFromWebPage.tf

```
resource "null_resource" "GetContentFromWeb" {
    triggers = {
        CurrentTime = timestamp()
    }

    # Getting Content From webpage
    provisioner "local-exec" {
        on_failure = continue
        command =  " curl http://${aws_instance.web.public_ip}/index.html"
    }

    # If webpage is configure then print the output
    depends_on = [null_resource.webpage-Configure]
}
```

The provided Terraform code defines a null_resource resource block named "GetContentFromWeb." Let's go through the code and explain each part:

Defining the null_resource Resource:

The code starts by declaring a null_resource named "GetContentFromWeb."

Triggers for Forcing Replacement:

The triggers block sets up a trigger with the CurrentTime attribute, which is set to the current timestamp using the timestamp() function. As seen in previous examples, this is used to force the null_resource to be recreated every time the Terraform configuration is applied. It ensures that the provisioner inside this resource is executed on every Terraform run.

Local Execution Provisioner:

The provisioner "local-exec" block defines a provisioner that will run a local command on the machine where Terraform is executed.

on_failure = continue: This attribute sets the behavior when the command execution fails. Here, it is set to "continue," which means if the command fails, Terraform will continue executing the rest of the configuration instead of terminating the process.

command: It specifies the command to be executed. In this case, the command uses curl to fetch the content of the "index.html" file from the EC2 instance with the public IP address aws_instance.web.public_ip. The curl command fetches the content of the web page.

Dependency on Another null_resource:

The depends_on attribute ensures that this null_resource is created only after another null_resource named "webpage-Configure" (not provided in the code snippet) is successfully created. The presence of this dependency suggests that the "GetContentFromWeb" provisioner should run after the "webpage-Configure" provisioner has completed its execution.

Keep in mind that the success or failure of the "curl" command execution will not impact the Terraform apply process due to the use of on_failure = continue. However, if the "webpage-Configure" provisioner is not successfully executed, this provisioner might not produce the expected results, as it relies on the availability of the "index.html" file on the EC2 instance.

Null_resource-EnvironmentDestroy.tf

```
resource "null_resource" "EnvironmentDestroy" {

    # Run the command when the Environment is Destroying
    provisioner "local-exec" {
        when = destroy
        command =  "echo Destorying Infrastructure"
    }
}
```

The code starts by declaring a null_resource named "EnvironmentDestroy." As previously mentioned, a null_resource is a resource that serves as a placeholder for executing actions or provisioners that don't directly manage infrastructure resources.

Local Execution Provisioner for Destroy:

The provisioner "local-exec" block is used to define a provisioner that will run a local command on the machine where Terraform is executed.

when = destroy: The when attribute specifies the condition under which the provisioner should run. In this case, it is set to "destroy," which means the provisioner will only execute when Terraform is destroying the infrastructure with the "terraform destroy" command.

command: It specifies the command to be executed. In this case, the command is "echo Destroying Infrastructure." The command is a simple example for demonstration purposes, and you can replace it with any command you wish to run during environment destruction.

Execution during Terraform Destroy:

The "EnvironmentDestroy" resource is triggered and executes the provisioner only when you run "terraform destroy" to destroy the Terraform-managed infrastructure. The provisioner's command will run on the local machine where Terraform is being executed.

The purpose of this null_resource and provisioner is to provide a mechanism for running additional commands or scripts when destroying the Terraform environment. This can be useful for performing cleanup tasks, logging, or any other custom actions you may need to take when tearing down infrastructure.

Keep in mind that the provisioner runs locally on the machine where Terraform is executed, and it does not directly affect the cloud resources managed by Terraform. The provisioner's commands are executed outside the scope of the Terraform state and cloud platform, so they won't be reflected in the Terraform state or affect the state of the cloud resources.

terraform.tfvars

```
# Variable file to make changes

# Default Security Group

mySecurityGroup="sg-010f683d9801ca435"

# Default Region
defaultRegion="ap-south-1"

# AMI ID
amiId="ami-072ec8f4ea4a6f2cf"

# Default Key
defaultKey="HimanshuTF"

# Default Instance Type
defaultInstanceType="t2.micro"

# Default Instance State
defaultInstanceState = "running"
```


The provided terraform.tfvars file is a variable file in Terraform that sets values for various variables used in the Terraform configuration. Let's explain each variable:

mySecurityGroup:

This variable sets the default security group ID that will be associated with the EC2 instances created in the Terraform configuration. The value "sg-010f683d9801ca435" is an example security group ID. You can replace it with the appropriate security group ID that exists in your AWS environment.

defaultRegion:

This variable sets the default AWS region where the resources will be provisioned. The value "ap-south-1" represents the Asia Pacific (Mumbai) region. You can change this value to your preferred AWS region.

amiId:

This variable sets the AMI (Amazon Machine Image) ID to use when launching the EC2 instances. The value "ami-072ec8f4ea4a6f2cf" is an example AMI ID. You should provide the correct AMI ID based on the AMI you want to use for your EC2 instances.

defaultKey:

This variable sets the default AWS key pair (SSH key) name that will be associated with the EC2 instances. The value "HimanshuTF" is an example key pair name. Make sure to specify the correct key pair name you have already created in your AWS account.

defaultInstanceType:

This variable sets the default EC2 instance type to use when launching the EC2 instances. The value "t2.micro" represents the instance type with a small amount of CPU and memory resources. You can change this value to a different instance type according to your requirements.

defaultInstanceState:

This variable sets the default desired state of the EC2 instances. The value "running" indicates that the instances should be in the running state. You can adjust this value to "stopped" or "terminated" if you want different states for the instances during provisioning.

By using a terraform.tfvars file, you can centralize and manage the values of these variables separately from the main Terraform configuration. When running terraform apply, Terraform will automatically load the values from this file and use them to configure the resources accordingly.

Make sure to keep the terraform.tfvars file secure and not include any sensitive information, as it may contain access credentials or other sensitive data used in the Terraform configuration.

The `**terraform refresh**` command is used to reconcile the Terraform state with the real-world resources managed by the provider. It is a subcommand of Terraform CLI that helps in updating the Terraform state file based on the current state of the infrastructure resources in the cloud.

When you run `terraform apply`, Terraform creates or updates the resources defined in your configuration to match the desired state. However, Terraform may not always have the latest state of the real-world resources in its state file, especially if resources are created or modified outside of Terraform. In such cases, running `terraform refresh` can be beneficial to bring the state file up to date.

Here's what `terraform refresh` does:

1. Reads Real-World Resource State:

The command communicates with the provider (e.g., AWS, Azure, GCP) to get the current state of all the resources managed by the configuration. It fetches information about the resources directly from the cloud platform.

2. Updates Terraform State File:

After fetching the latest state from the provider, Terraform updates its local state file (`terraform.tfstate`) to match the real-world resource state. This ensures that Terraform has an accurate representation of the current infrastructure.

3. No Resource Creation or Deletion:

Unlike `terraform apply`, `terraform refresh` does not create or modify any resources. It is a read-only operation focused on updating the state file without making changes to the actual resources.

Running `terraform refresh` is generally useful in the following scenarios:

- When resources are created or modified outside of Terraform (e.g., manually in the cloud console or using other tools).
- After importing existing resources into Terraform, ensuring the state file reflects the real-world state.
- As a safety measure before performing destructive operations with `terraform apply`, to avoid accidental modifications.

Keep in mind that `terraform refresh` can be resource-intensive, as it fetches the latest state of all resources from the cloud platform. Therefore, it is not typically needed as part of regular Terraform workflows but can be handy in specific situations where you need to synchronize the state file with the actual infrastructure.