PART1

Saturday, December 31, 2022 1:51 PM

```
touch ~/.bashrc

terraform -install-autocomplete

# Configure the Azure provider
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm" #registry.terraform.io/hashicorp/azurerm
      version = "~> 3.0.2"
    }
  }

  required_version = ">= 1.1.0"
}

provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "rg" {
  name     = "myTFResourceGroup"
  location = "westus2"
}
```

Use terraform show again to see the new values associated with this resource group.
Run terraform state list to get the updated list of resources managed in your workspace.

After you define the outputs.tf file with required outputs the outputs will be displayed in the cli after tf apply
terraform output outputvar to view that particular value

```
terraform providers lock \
  -platform=linux_arm64 \
  -platform=linux_amd64 \
  -platform=darwin_amd64 \
  -platform=windows_amd64
```

The above command will download and verify the official packages for all of the required providers across all four of the given platforms, and then record both zh: and h1: checksums for each of them in the lock file, thus avoiding the case where Terraform will learn about a h1: equivalent only at a later time

do not recommend using any provisioners except the built-in file, local-exec, and remote-exec provisioners.

```
resource "aws_instance" "web" {
  # ...

  provisioner "local-exec" {
    command = "echo The server's IP address is ${self.private_ip}"
  }
}
 on_failure = continue
 when    = destroy argument in provisioner says it to run when it is being deleted
```

The file provisioner copies files or directories from the machine running Terraform to the newly created resource.
The file provisioner supports both ssh and winrm type connections.

```
resource "aws_instance" "web" {
  # ...

  # Copies the myapp.conf file to /etc/myapp.conf
  provisioner "file" {
    source      = "conf/myapp.conf"
    destination = "/etc/myapp.conf"
  }

  # Copies the string in content into /tmp/file.log
  provisioner "file" {
    content     = "ami used: ${self.ami}"
    destination = "/tmp/file.log"
  }

  # Copies the configs.d folder to /etc/configs.d
  provisioner "file" {
    source      = "conf/configs.d"
    destination = "/etc"
  }
```

```
 # Copies all files and folders in apps/app1 to D:/IIS/webapp1
 provisioner "file" {
   source      = "apps/app1/"
Content = "direct content instead of specying the source"
   destination = "D:/IIS/webapp1"
 }
}
```

The local-exec provisioner invokes a local executable after a resource is created. This invokes a process on the machine running Terraform, not on the resource.

```
resource "aws_instance" "web" {
 # ...

 provisioner "local-exec" {
   command = "echo ${self.private_ip} >> private_ips.txt"
 }
}
When=delete , working_dir =  "." ,interpreter = ["perl", "-e"]
```

The remote-exec provisioner invokes a script on a remote resource after it is created.(ssh,winrm)

```
resource "aws_instance" "web" {
 # ...

 # Establishes connection to be used by all
 # generic remote provisioners (i.e. file/remote-exec)
 connection {
   type     = "ssh"
   user     = "root"
   password = var.root_password
   host     = self.public_ip
 }

 provisioner "remote-exec" {
   inline = [
     "puppet apply",
     "consul join ${aws_instance.web.private_ip}",
   ]
 }
}
resource "aws_instance" "web" {
 # ...

 provisioner "file" {
   source      = "script.sh"
   destination = "/tmp/script.sh"
 }

 provisioner "remote-exec" {
   inline = [
     "chmod +x /tmp/script.sh",
     "/tmp/script.sh args",
   ]
 }
}
Script = "pathtoscript" , scripts = "paths to scripts location"
```

```
resource "aws_instance" "cluster" {
 count = 3

 # ...
}

resource "null_resource" "cluster" {
 # Changes to any instance of the cluster requires re-provisioning
 triggers = {
   cluster_instance_ids = "${join(",", aws_instance.cluster.*.id)}"
 }

 # Bootstrap script can run on any instance of the cluster
 # So we just choose the first in this case
 connection {
   host = "${element(aws_instance.cluster.*.public_ip, 0)}"
 }

 provisioner "remote-exec" {
   # Bootstrap script called with private_ip of each node in the cluster
   inline = [
     "bootstrap-cluster.sh ${join(" ", aws_instance.cluster.*.private_ip)}",
   ]
 }
}
```

Terraform state rm "vm_instance.name" to remove it from the state config file
Credentials can be specified in the connection argument in providers
Remote backends:

- Local
- Azurerm
- Consul
- Cos
- Gcs
- Http
- Oss
- Pg
- S3

For azure remote backedn

```
Terraform{
Backend "azurerm"{
Resource_group_name = "name"
Storage_account_name  = "name"
Container_name = "tfstate"
Key="prod.terraform.tfstate"
}
}
```

Meta_Arguments:

- Depends_on
- Count
- For_each
- Provider
- Lifecycle
- Provisioner

We can specify -lock in the terraform command to explicilty lock the state writes
If automatic unlocking failed then we can use the force_unlock command

Locals:

```
Locals{
Key1= "value1"
Key2 = "value2"
}
Locals{
Comman_tags={
Service = local.Key1
Owner = local.owner
}
}
```

```
terraform plan -replace="aws_instance.example"

terraform state mv -state-out=../terraform.tfstate aws_instance.example_new aws_instance.example_new

terraform import aws_security_group.sg_8080 $(terraform output -raw security_group)
```

Data sources:

- Data "resource_name(aws_ami)" "ami"{
  Filter={
  "image","ddd
  dca
  }
  }
- It is specific to the provider and it can be used to collect data outisde of terraform

```
data "aws_ami" "amazon_linux" {
 most_recent = true
 owners      = ["amazon"]

 filter {
  name   = "name"
  values = ["amzn2-ami-hvm-*-x86_64-gp2"]
 }
}
```

Built in functions:
(here there is a file where it contains the userdata for ec2 with ${varname} in it

- resource "aws_instance" "web" {
  ami   = data.aws_ami.ubuntu.id
  instance_type   = "t2.micro"
  subnet_id   = aws_subnet.subnet_public.id
  vpc_security_group_ids = [aws_security_group.sg_8080.id]
  associate_public_ip_address = true
  user_data   = templatefile("user_data.tftpl", { department = var.user_department, name =
  var.user_name })
  }
- variable "aws_amis" {
  type = map
  default = {
  "us-east-1" = "ami-0739f8cdb239fe9ae"
  "us-west-2" = "ami-008b09448b998a562"
  "us-east-2" = "ami-0ebc8f6f580a04647"
  }
  }
  resource "aws_instance" "web" {
  - ami   = data.aws_ami.ubuntu.id
  + ami   = lookup(var.aws_amis, var.aws_region)
  instance_type   = "t2.micro"
  subnet_id   = aws_subnet.subnet_public.id
  vpc_security_group_ids = [aws_security_group.sg_8080.id]
  associate_public_ip_address = true
  user_data   = templatefile("user_data.tftpl", { department = var.user_department, name =
  var.user_name })
  }
- resource "aws_key_pair" "ssh_key" {
  key_name = "ssh_key"
  public_key = file("ssh_key.pub")
  }

Conditional expressions:

- count   = (var.high_availability == true ? 3 : 1)
- tags   = merge(local.common_tags)
- Splat expressions =
  output "private_addresses" {
  description = "Private DNS for AWS instances"
  value   = aws_instance.ubuntu[*].private_dns
  }

We can use a vault provider and speicfy the url where the vault runs and the authenticatio options and the secret and give the secret values

You can set TF_LOG to one of the log levels (in order of decreasing verbosity) TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs.
Logging can be enabled separately for terraform itself and the provider plugins using the TF_LOG_CORE or TF_LOG_PROVIDER environment variables. These take the same level arguments as TF_LOG, but only activate a subset of the logs.

To persist logged output you can set TF_LOG_PATH in order to force the log to always be appended to a specific file when logging is enabled. Note that even when TF_LOG_PATH is set, TF_LOG must be set in order for any logging to be enabled.

Terraform Cloud uses Sentinel as part of Teams & Governance to enable granular policy control for your infrastructure. Sentinel is a language and policy framework, which restricts Terraform actions to defined, allowed behaviors. Policy authors manage Sentinel policies in Terraform Cloud with policy sets, which are groups of policies. Organization owners control the scope of policy sets by applying certain policy sets to the entire organization or to select workspaces.

The Policy-as-Code framework enables you to treat your governance requirements as you would your applications: written by operators, controlled in VCS, reviewed, and automated during your deployment process.

Sentinel uses the four Terraform Cloud imports to define policy rules: plan, configuration, state, run.

tfplan - This provides access to a Terraform plan, the file Terraform creates as a result of a plan. The plan data represent the changes that Terraform needs to make to infrastructure to reach the desired state represented by the configuration.
tfconfig - This provides access to data describing a Terraform configuration, the set of ".tf" files that you write to describe the desired infrastructure state.
tfstate - This provides access to data describing the Terraform state, the file Terraform uses to map real-world resources to your configuration.
tfrun - This provides access to data associated with a run in Terraform Cloud, such as the run's workspace.

PART2

Saturday, December 31, 2022 1:51 PM

1) Scope(confirm the resources required for a project)
2) Author(author the config files based on the4 scope)
3) Initialize(download the provider plugins and initialie directory)
4) Plan(view execution plan for resources created,modified,destroyed)
5) Apply(create actual infrastructure resources)

Versions available:
- Terraform open source(only CLI,no concurrent deployments)
- Terraform Cloud(CLI,GUI,concurrent deployments)
- Terraform enterprise(GUI,CLI, conurrent deployments,secure deployments)

Authentication :
- Using gcloud sdk
- Using service account in vm's
- Using env vars for service account keys in onprem

Example workflow:
- Create .tf files
  Resource "google_compute_network" "my_network"{
  Name="my_vpc"
  }
- Terraform init
- Terraform plan
- Terraform apply
- Terraform destroy
- Terraform fmt(autoformat to match canonical conventions)

Need to organize .tf files in terms of directories and files for each functions like main.tf, providers.tf, variables.tf, outputs.tf

In the main.tf the actual resource configs are written

In the providers.tf the cloud provider plugin is downloaded and given configuration options like project id ,etc..

In the variables.tf we define the variables that looks into a terraform.tfvars for runtime stuff like
Variable location{
..
} in variables.tf and
Localtion=''US'' in terraform.tfvars

In the outputs.tf we can define outputs in order to retrieve the created resources config url or similar to bucket url
Output "buckety_URL" {
    Value = google_storage_bucket.mybucket.URL
}

In the terraform.tfstate stores the state for the resources we create using terraform locally,remotely

Modules:

- Is a set of terraform config files in a single directory

For running the terraform validator we use gcloud beta terraform vet

When developing code we tend to use the resource attribute's values
For that we can use **resourcename.resourcetype.attribute**

Meta-arguments:

- To customize the behaviour of resources

1) Count -> create multiple instances
   {
   Count=3
   Name = "devVM${count.index + 1}"
   }
2) For_each -> create multiple resource instances as per a set of strings
   {
   For_each = toset(["us-central1-a","asia-east-b","europe-westr-a"]}
   Name = "dev-${ech.value}"
   Zone = each.value
   }
3) Depends_on -> specify explicit dependency
4) Lifecycle -> define lifecycle of a resource

Dependencies:

- Implicit dependency(dependencies known to terraform are detected automatically)
  When we give the **resourcename.resourcetype.attribute** inside the resource definition to take the value from other resource defined in the same file
- Explicit dependencies should be configured explictly and are not known by default)
  Depends_on = [google_compute_instances.server]

Variables:

- Should specify in variables.tf
  Variable "name" {
  Type= string
  Description = "noting"
  Default = "US"
  Sensitive = true # this will not be shown in the terraform plam,apply commands ops
  }
- in the main.tf we can reference it by "${var.instance_name}"
- Also we can alternatively pass variables in .tfvars file and pass it through (tf apply -var-file myvars.tfvars)
- Tf apply -var project_id="projectid"
- TF_VAR_project_id="projectid" tf apply
- Tf apply (if using terraform.tfvars)
- In the terraform.tfvars we can specify variables as a line (key=val)
- If no value is specified then terraform asks when applying

Validation{
Conditoin = containsn(["","",""],var.varname)
Error_message = "Ddddd"
}
(

```
variable "location" {
  description = "The Azure Region in which all resources in this example
should be provisioned"
```

```
 validation{
condition = contains(["ram","shankar","is"],var.location)
error_message = "Ddddd"
}
}

)
```

- For validation

Outputs:

- Usually in outputs.tf file

Output "picture_URL" {
Description: "something"
Value = google_storage_bucket.picture.self_link
Sensitive=true
}

Terraform registry:

- Interactive resource for discovering providers,modules
- Solutions developed by hashicorp,thirdparty etc..

Cloud foundation kit:

- Provides a series of reference modules for terraform that reflect gcp best pratices
- Modules in this can be used without modification to quickly build a repeatable enterprise ready foundation in gcp

terraform graph | dot -Tsvg > graph.svg(to view dependency graph)

Modules:

Module <name> {
Source = "./dir" (can be terraform registry,github,bitbucket,http urls,gcs bucket)
}
Also we can give the module a attribute given at runtime like
Module "" {
Source = ""
Network_name = module.my_network.network_name
}

Terraform states:
We can add a backend.tf file to store the state file remotely instead of locally

Terraform {
Backend "gcs" {
Bucket = "name"
Prefix = "terraform/state"
}}
terraform init -migrate-state
terraform refresh

terraform show - to show the state

terraform plan -out static_ip (to save the plan for future use the same)
Terraform apply "static_ip" (for using the same plan outputted)

terraform taint google_compute_instance.vm_instance (to tell terraform to recreate the instance)
terraform taint module.instances.google_compute_instance.tf-instance-461793

```
resource "google_compute_instance" "vm_instance" {
  name         = "terraform-instance"
  machine_type = "f1-micro"
  tags         = ["web", "dev"]
  provisioner "local-exec" {
    command = "echo ${google_compute_instance.vm_instance.name}:
${google_compute_instance.vm_instance.network_interface[0].access_config[0].nat_ip} >>
ip_address.txt"
  }
  # ...
}
```
(we can use provisioners like this to exec some commands)

In the module directory we will have the .tf files with variables (parameterized)
When we call the modules with module{} block we specify some attributes inside this block thjose are variables used in the modules

After defining module in the main.tf file we have to run terraform import module.modulename.resourcetype.resourcename to import the existing resources that are already available in gcp

Like 2 instances already available should be declared inside a directory/module and in the main tf file initialize the module with source as the director and run the command accordingly