Terraform notes

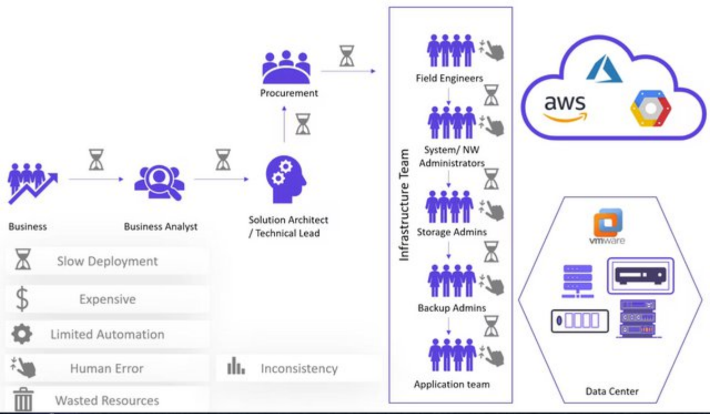
FOR DEVOPS ENGINEERS

Terraform Short Notes

Introduction

- Infrastructure as code (IaC) tools allow you to manage infrastructure with configuration files rather than through a graphical user interface. IaC allows you to build, change, and manage your infrastructure in a safe, consistent, and repeatable way by defining resource configurations you can version, reuse, and share.

Challenges in IT Infrastructure



Terraform

- Terraform is HashiCorp's infrastructure as a code tool. It lets you define resources and infrastructure in human-readable, declarative configuration files and manages your infrastructure's lifecycle. Using Terraform has several advantages over manually managing your infrastructure:
- Terraform can manage infrastructure on multiple cloud platforms.
- The human-readable configuration language helps you write infrastructure code quickly.
- Terraform's state allows you to track resource changes throughout your deployments.
- You can commit your configurations to version control to safely collaborate on infrastructure.

Installation

Linux

```
• sudo apt-get update && sudo apt-get install -y gnupg software-
  properties-common
• wget -O- https://apt.releases.hashicorp.com/gpg | \ gpg
   --dearmor | \
   sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg

• gpg --no-default-keyring \
   --keyring /usr/share/keyrings/hashicorp-archive-keyring.gpg \
   --fingerprint

• echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-
  keyring.gpg] \
   https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \ sudo
   tee /etc/apt/sources.list.d/hashicorp.list

• sudo apt update
• sudo apt-get install terraform
```

HCL

- Hashicorp Configuration Language, This low-level syntax of the Terraform language is defined in terms of a syntax called HCL, which is also used by configuration languages in other applications, and in particular other HashiCorp products. It is not necessary to know all of the details of HCL syntax in order to use Terraform, just knowing the basics, should be enough.

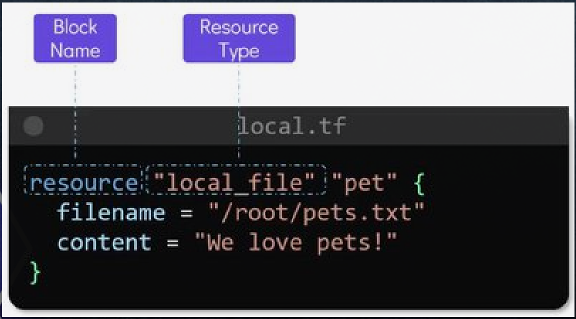
local.tf

```
<block> <parameters> {
   key1 = value1
   key2 = value2
}
```

- The Terraform language syntax is built around two key syntax constructs: arguments and blocks.

Blocks and Arguments

- A block is a container for other content and An argument assigns a value to a particular name:
- filename = "/home/ubuntu/abc123.txt"
- The identifier before the equals sign is the argument name, and the expression after the equals sign is the argument's value.



- Resource block: block name used to mention the type of the block. The resource block expects two labels, which are local_file and “pet” in the example above. A particular block type may have any number of required labels, or it may require none.

```
resource "<provider>_<resource type>" "<resource name>"
{ Argument1 = ""
Argument2=""
}
```

- Local = provider, file = type, “pet” name of the resource. Then
- we have arguments, filename, content, etc
- We can have multiple resources

```
resource "random_string" "rand-str"
{ length = 16
special = true
override_special = "!#$%&*()-_=+[]{}<>:?"
 }

output "rand-str" {
  value = random_string.rand-str[*].result
}
```

Execution of Infrastructure

Init-> plan-> apply terraform

init

This command will scan your tf files in that folder and install all the required automation things.

terraform plan

This command will create an execution plan for terraforming, the things that will be installed, the names, and the properties added.

terraform apply

The actual execution and automation happen in this command.

Terraform with Docker

- Terraform needs to be told which provider to use in the automation, hence we need to give the provider name with source and version.
- For Docker, we can use this block of code in your main.tf

Terraform block

```
terraform
  { required_providers
  {
    docker = {
      source = "kreuzwerker/docker"
      version = "~> 2.21.0"
        }
    }
```

Provider

- The provider block configures the specified provider, in this case, docker. A provider is a plugin that Terraform uses to create and manage your resources.
- provider "docker" {}

Resource

- Use resource blocks to define components of your infrastructure. A resource might be a physical or virtual component such as a Docker container, or it can be a logical resource such as a Heroku application.
- Resource blocks have two strings before the block: the resource type and the resource name. In this example, the first resource type is docker_image and the name is Nginx.

```
esource "docker_image" "nginx"
 { name   = "nginx:latest"
 keep_locally = false
}

resource "docker_container" "nginx-ctr"
 { image = docker_image.nginx.latest
 name = "tutorial-shubham"
 ports {
   internal = 80
   external = 80
 }
}
```

Note: In case Docker is not installed

```
sudo apt-get install docker.io
sudo docker ps
sudo chown $USER /var/run/docker.sock
```

more Terraform commands:

```
<|'
- terraform fmt
- terraform validate
- terraform show
- terraform state list
```

Terraform Variables

We can create a variables.tf file which will hold all the variables.

```
<|'

variable "filename" {
default = "/home/ubuntu/terrform-tutorials/terraform-variables/demo-var.txt"
}

variable "content" {
default = "This is coming from a variable which was updated"
}

These variables can be accessed by var object in main.tf
resource "local_file" "devops" {
filename = var.filename
content = var.content
}
```

Data Typ es in Terraform

```
<|'
variable "file_contents"
{ type = map
default = {
"statement1" = "this is cool"
"statement2" = "this is cooler"
}
}
```

List

variables.tf

```
variable "prefix" {
  default = ["Mr", "Mrs", "Sir"]
  type = list(string)
}
```

variables.tf

```
variable "prefix" {
  default = [   1 , 2 , 3   ]
  type = list(number)
}
```

Set

```
variables.tf

variable "prefix" {
  default = ["Mr", "Mrs", "Sir"]
  type = set(string)
}
```

```
variables.tf

variable "fruit" {
  default = ["apple", “banana"]
  type = set(string)
}
```

```
variables.tf

variable "age" {
  default = [    10 , 12 , 15     ]
  type = set(number)
}
```

Object

```
variable "devops" {
          type =
          object({ name =
          string
          items = list(number)
})

          default = {
          name = "shubham"
          items = [1,2,3,4]
}
```

Use terraform refresh

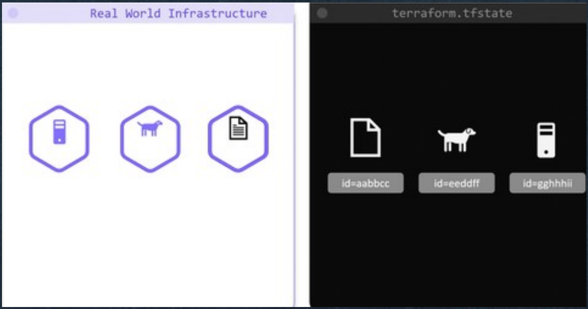
To refresh the state of your configuration file, reload the variables

```
ubuntu@ip-172-31-88-170:~/terrform-tutorials/terraform-variables$ terraform refresh
local_file.devops: Refreshing state... [id=2c0ce58b726a9a7548c1369aa2c2fd71f4a445c5]
local_file.devops-new: Refreshing state... [id=60f51187e76a9de0ff3df31f051bde04da2da891]

Outputs:

devops-items = tolist([
  1,
  2,
  3,
  4,
])
devops-op = "shubham"
ubuntu@ip-172-31-88-170:~/terrform-tutorials/terraform-variables$ terraform output
devops-items = tolist([
  1,
  2,
  3,
  4,
])
```

Terraform State

- Whenever we do terraform init, the plugins are installed
- Whenever we do a terraform plan, the execution plan is generated
- Whenever we do terraform apply, the execution is done and state is maintained

- If we don’t have state we can still run the above commands, but state is useful to keep a record of why and how infrastructure was created at the first place.
- State is like a blueprint of the Real-world Infrastructure with some unique IDs and attributes.



- Used to improve performance, dependency management, ect

Terraform with AWS

- Provisioning on AWS is quite easy and straightforward with Terraform.

Prerequisites

AWS CLI installed (Done)

- The AWS Command Line Interface (AWS CLI) is a unified tool to manage your AWS services. With just one tool to download and configure, you can control multiple AWS services from the command line and automate them through scripts.

AWS IAm user (Done)

- IAm (Identity Access management) AWS Identity and Access management (IAm) is a web service that helps you securely control access to AWS resources. You use IAm to control who is authenticated (signed in) and authorized (has permissions) to use resources.

- In order to connect your AWS account and Terraform, you need the access keys and secret access keys exported to your machine.

```
export AWS_ACCESS_KEY_ID=<access key>
export AWS_SECRET_ACCESS_KEY=<secret access key>
```

Install required providers

```
terraform
 { required_providers
 {
        aws = {
        source = "hashicorp/aws" version
        = "~> 4.1G"
}
}
        required_version = ">= 1.2.0"
}
Add the region where you want your instances to be
provider "aws" {
region = "us-east-1"
```

AWS EC2 instance provisioning

```
resource "aws_instance" "aws_ec2_test"
        { count = 4
        ami = "ami-08c40ec9ead489470"
        instance_type = "t2.micro" tags
        = {
     Name = "TerraformTestServerInstance"
  }
}
```

Get the Public IPs for the provisioned instances

```
output "instance_pub_ip" {
        value = aws_instance.aws_ec2_test[*].public_ip
}
```

AWS S3

- For s3, the bucket name should be unique

```
resource "aws_s3_bucket" "b" {
  bucket = "trainwithshubham-tf-test-bucket"

  tags = {
    Name        = "trainwithshubham-bucket"
    Environment = "Dev"
  }
}
```

Terraform State commands

- terraform state list: List resources within terraform state.
- terraform-state mv : move items within terraform state. This will be used to resource renaming without destroy, apply command.
- terraform state pull: manually download and output the state from the state file.
- terraform state push: manually upload a local state file to the remote state.
- terraform state rm: Remove items from the state. Items removed from the state are not physically destroyed. This item no longer managed by Terraform.
- terraform state show: Show attributes of a single resource in the state.

States

- Terraform uses state to keep track of the infrastructure it manages. To use Terraform effectively, you must keep your state accurate and secure.

- State is a necessary requirement for Terraform to function. It is often asked if it is possible for Terraform to work without state, or for Terraform to not use state and just inspect cloud resources on every run.

- Terraform requires some sort of database to map Terraform config to the real world. Alongside the mappings between resources and remote objects, terraform must also track metadata such as resource dependencies. Terraform stores a cache of the attribute values for all resources in the state. This is done to improve performance.

- For small infrastructures, terraform can query your providers and sync the latest attributes from all your resources. This is the default behaviour of Terraform: for every plan and application, terraform will sync all resources in your state.

- For larger infrastructures, querying every resource is too slow. Larger users of Terraform make heavy use of the -refresh=falseflag as well as the -targetflag to work around this. In these scenarios, the cached state is treated as the record of truth.

State Locking

- State locking happens automatically on all operations that could write state. You won't see any message that it is happening. If state locking fails, terraform will not continue. You can disable state locking in most commands with the -lock flag but it is not recommended.

- Terraform has a force-unlock command to manually unlock the state if unlocking failed.

Syntax

```
terraform force-unlock [options] LOCK_ID [DIR]
```

Sensitive Data

- Terraform state can contain sensitive data, e.g. database password, etc. When using a remote state, the state is only ever held in memory when used by Terraform.
- The S3 backend supports encryption at rest when the encrypt option is enabled. IAm policies and logging can be used to identify any invalid access. Requests for the state go over a TLS connection.

Note

- Setting an output value in the root module as sensitive prevents Terraform from showing its value in the list of outputs at the end of terraform apply. However, output values are still recorded in the state and so will be visible to anyone who is able to access the state data.

```
output "db_password" {
value = aws_db_instance.db.password description
= "The password for logging
                 in to the database."
sensitive = true
}
```

Backend management

- A backend in Terraform determines how state is loaded and how an operation such as apply is executed
- Terraform must initialize any configured backend before use.

Local

- By default, terraform uses the “local” backend. After running first terraform apply the terraform.tfstate file created in the same directory of main.tf
- terraform.tfstate file contains JSON data.
- The local backend stores state on the local filesystem, locks the state using system APIs, and performs operations locally

```
terraform
{ backend "local" {
path = "relative/path/to/terraform.tfstate"
}
}
```

Remote

- When working with Terraform in a team, the use of a local file makes Terraform usage complicated because each user must make sure they always have the latest state data before running Terraform and make sure that nobody else runs Terraform at the same time.
- With a remote state, terraform writes the state data to a remote data store, which can then be shared between all members of a team.

```
terraform {
backend "remote" {}
}
```

This is called partial configuration.

- When configuring a remote backend in Terraform, it might be a good idea to purposely omit some of the required arguments to ensure secrets and other relevant data are not inadvertently shared with others.

```
terraform init -backend-config=backend.hcl
```

modules

- A module is a simple directory that contains other .tf files. Using modules we can make the code reusable. modules are local or remote.

Calling Child module

- Input variables to accept values from the calling module. Output values to return results to the calling module, which it can then use to populate arguments elsewhere. Resources to define one or more infrastructure objects that the module will manage.

```
variable "image_id"
{ type = string
}resource "aws_instance" "myec2"
{ ami = var.image_id
instance_type = "t2.micro"
}
output "instance_ip_addr" {
value = aws_instance.myec2.private_ip
}
module "dbserver"
{ source = "./db"
image_id = "ami-0528a5175983e7f28"
}
```

- module outputs are very similar to module inputs, an example in a module output:

```
output "privateip" {
value = aws_instance.myec2.private_ip
}
```

- It is recommended to explicitly constraining the acceptable version numbers for each external module to avoid unexpected or unwanted changes.
- Version constraints are supported only for modules installed from a module registry, such as the Terraform Registry or Terraform Cloud's private module registry.

Debugging in Terraform

- Terraform has detailed logs that can be enabled by setting the TF_LOG environment variable to any value.
- You can set TF_LOG to one of the log levels TRACE, DEBUG, INFO, WARN or ERROR to change the verbosity of the logs.
- export TF_LOG=TRACE
- To persist logged output, you can set TF_LOG_PATH
- TF_LOG_PATH=./terraform.log

Terraform Functions

- The Terraform language includes a number of built-in functions that you can use to transform and combine values
- max(5, 12, 9)
  12
- The Terraform language does not support user-defined functions, and so only the functions built into the language are available for use

- into the language are available for use
- Some other example built-in functions
- element retrieves a single element from a list
  element(["a", "b", "c"], 1)
  b

- lookup retrieves the value of a single element from a map, given its key
  lookup({a="ay", b="bee"}, "c", "what?")
  what?

Count and Count Index

- The count parameter on resources can simplify configurations and let you scale resources by simply incrementing a number.

- In resource blocks where the count is set, an additional count object (count.index) is available in expressions, so that you can modify the configuration of each instance.

```
resource "aws_instance" "myec2"
{ ami = var.image_id
instance_type = "t2.micro"
count = 3
}
output "instance_ip_addr" {
value = aws_instance.myec2[*].private_ip
}
```

```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

instance_ip_addr = [
  "172.31.41.113",
  "172.31.35.152",
  "172.31.45.143",
]
```

Provisioners

- Provisioners can be used to model specific actions on the local machine or on a remote machine to prepare servers or other infrastructure objects for service. Provisioners are inside the resource block. Note: Provisioners should only be used as a last resort. For most common situations there are better alternatives.

file Provisioner

- The file provisioner is used to copy files or directories from the machine executing Terraform to the newly created resource.

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

local-exec Provisioner

- The local-exec provisioner requires no other configuration, but most other provisioners must connect to the remote system using SSH or WinRm.

```
resource "aws_instance" "web"
{ # ...
provisioner "local-exec" {
command = "echo The server's IP address is
${self.private_ip}"
}
}
```

remote-exec Provisioner

- The remote-exec provisioner invokes a script on a remote resource after it is created. This can be used to run a configuration management tool, bootstrap into a cluster, etc.

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

Creation-time Provisioners

- By default, provisioners run when the resource they are defined within is created. Creation-time provisioners are only run during creation, not during updating or any other lifecycle. They are meant to perform bootstrapping of a system. If a creation-time provisioner fails, the resource is marked as tainted. A tainted resource will be planned for destruction and recreation upon the next terraform apply.

Destroy-time Provisioners

- if when = “destroy” is specified, the provisioner will run when the resource it is defined within is destroyed.

Terraform with AWS

Setup an AWS user for use with Terraform

- We now need to create an AWS user that we can use with Terraform. We are going to create an account which has administrator permissions.

1. Log into your AWS account and you have access and go to the IAm section, you can do this by searching for IAm in the search box on the main AWS page and then clicking on the link
2. 2 Select Users from the left-hand menu
3. Select Add User at the top
4. Type in any username you like
5. For access type select Programmatic access only
6. Click Next
7. On the set permissions screen select.

Attach existing policies direct

1. Tick AdministratorAccess which should be the top of the list
2. Click Next
3. Click Next again, now you should see a summary of the user you are about to create
4. Click the Create User button and the user should be created
5. Store the Access Key ID and Secret Access Key somewhere safe as this is the only time you will see them

Setup an AWS Credentials file

- The last thing we need to do is create an AWS Credentials file. This is so that Terraform can get the programmatic credentials for the AWS user we created above.

- You need to create a file and with the following text, replacing the two placeholders with the access key id and secret access key you got from AWS when you created your admin user.
- 1 [default]
- 2 aws_access_key_id = <access_key_id_here>
- 3 aws_secret_access_key = <secret_access_key_here>
- Lastly save the file to the path given in the table below based on your OS:

OS	Credentials file path
Windows	%UserProfile%/.aws/credentials
Mac OS/Linux	~/.aws/credentials

meta-Arguments in Terraform

- Count
- for_each
- depends_on

Count

```
terraform
 { required_providers
 {
        aws = {
        source = "hashicorp/aws"
        version = "~> 4.16"
}
}
        required_version = ">= 1.2.0"
}
provider "aws"
{ region = "us-east-
1"
}

resource "aws_instance" "server"
  { count = 4
```

```
ami           = "ami-08c40ec9ead489470"
instance_type = "t2.micro"

tags = {
  Name = "Server ${count.index}"
}
}
```

for_each

```
terraform
 { required_providers
 {
        aws = {
        source = "hashicorp/aws"
        version = "~> 4.1G"
}
}
        required_version = ">= 1.2.0"
}

provider "aws"
{ region = "us-east-
1"
}
locals {
  ami_ids = toset([
    "ami-0b0dcb50G7f052aG3",
    "ami-08c40ec9ead489470",
  ])
}

resource "aws_instance" "server"
  { for_each = local.ami_ids

  ami           = each.key
  instance_type = "t2.micro"

  tags = {
    Name = "Server ${each.key}"
  }
```

```
}
```

multiple key_value iteration

```
locals
  { ami_ids = {
   "linux" :"ami-0b0dcb50G7f052aG3",
   "ubuntu": "ami-08c40ec9ead489470",
  }
}

resource "aws_instance" "server"
  { for_each = local.ami_ids

  ami       = each.value
  instance_type = "t2.micro"

  tags = {
    Name = "Server ${each.key}"
  }
}
```

Thank You