**infrastrucre?**

Infrastructure is the collection of hardware and software elements such as computing power, networking, and storage which are required to enable cloud computing**.**

**IaC?**

Infrastructure as code (IAC) means writing the code to create & manage infrastructure

Benefits of IaC

Speed

it uses automation

Consistency

Since it is code, it generates the same result every time. It provisioned the same

environment every time, enabling improved infrastructure consistency at all times.

Cost

One of the main benefits of IAC is, without a doubt, lowering the costs of infrastructure

management. With everything automated and organized, engineers save up on time and

cost which can be wisely invested in performing other manual tasks

Minimum Risk

IaC allows server configuration that can be documented, logged, and tracked later for

reference. Configuration files will be reviewed by a person or policy as a code (sentinel)

for security leakages.

Everything Codified

The main benefit of IaC is explicit coding to configure files in use. You can share codes

with the team, test them to ensure accuracy, maintain uniformity and update your

infrastructure into the same flow of IaC.

Version Controlled, Integrated

Since the infrastructure configurations are codified, we can check-in into version control

like GitHub and start versioning it.

IAC allows you to track and give insight on what, who, when, and why anything changed

in the process of deployment. This has more transparency which we lack in traditional

infrastructure management.

Terraform

* Platform Agnostic
* State Management.
* Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently.
* A provider is responsible for understanding API interactions and exposing resources.
* Terraform can support for multiple providers like aws, azure, google.
* It is a one single tool which automates the entire infrastructure whether its cloud or on premise.
* To make a provider available on Terraform, we need to make a terraform init, these

commands download any plugins we need for our providers. If for example, we need to

copy the plugin directory manually, we can do it, moving the files to .terraform.d/plugins

provider "aws" {

region = "us-east-1"

}

If the plugin is already installed, terraform init will not download again unless to

upgrade the version, run terraform init -upgrade.

What is the Terraform life cycle?

Every resource that is managed by Terraform has a lifecycle, this lifecycle contains three stages; Apply (Create), Update, and Destroy.

What is Idempotent in terms of IaC?

The idempotent characteristic provided by IaC tools ensures that, even if the same code is applied multiple times, the result remains the same.

What are Day 0 and Day 1 activities?

“Day 0” code provisions and configures your initial infrastructure.

“Day 1” refers to OS and application configurations you apply after you’ve initially built your infrastructure.

What are the use cases of Terraform?

Resource Schedulers

Multi-Cloud Deployment

Where do you describe all the components or your entire datacentre so that Terraform provision those?

Configuration files ends with \*.tf

How can Terraform build infrastructure so efficiently?

Terraform builds a graph of all your resources, and parallelizes the creation and modification of any non-dependent resources.

What is multi-cloud deployment?

Provisioning your infrastructure into multiple cloud providers to increase fault-tolerance of your applications.

How multi-cloud deployment is useful?

By using only a single region or cloud provider, fault tolerance is limited by the availability of that provider. Having a multi-cloud deployment allows for more graceful recovery of the loss of a region or entire provider.

What is cloud-agnostic in terms of provisioning tools?

cloud-agnostic and allows a single configuration to be used to manage multiple providers, and to even handle cross-cloud dependencies.

How using IaC make it easy to provision infrastructure?

IaC makes it easy to provision and apply infrastructure configurations, saving time. It standardizes workflows across different infrastructure providers (e.g., VMware, AWS, Azure, GCP, etc.) by using a common syntax across all of them.

Multiple Providers

You can optionally define multiple configurations for the same provider and select which

one to use on a per-resource or per-module basis.

#default configuration

provider "aws" {

region = "us-east-1"

}

# reference this as `aws.west`.

provider "aws" {

alias = "west"

region = "us-west-2"

}

Versioning..

Terraform current version v1.3.9,v1.4.6,v1.5.3

Aws cli version 2.12

The required version setting can be used to constrain which version of the Terraform

CLI can be used with your configuration. If the running version of Terraform doesn't

match the constraints specified, Terraform will produce an error and exit without taking

any further actions.

terraform {

required\_version = ">= 1.3.9"

}

The value for required\_version is a string containing a comma-separated list of

constraints. Each constraint is an operator followed by a version number, such as >

0.12.0. The following constraint operators are allowed:

• = (or no operator): exact version equality

• !=: version not equal

• >, >=, <, <=: version comparison, where “greater than” is a larger version

number

• ~>: pessimistic constraint operator, constraining both the oldest and newest

version allowed. For example, ~> 0.9 is equivalent to >= 0.9, < 1.0, and ~>

0.8.4, is equivalent to >= 0.8.4, < 0.9

We can also specify a provider version requirement

provider "aws" {

region = "us-east-1"

version = ">= 2.9.0"

}

Show version: $ terraform version

Terraform Workflow

Terraform Init

* The terraform init command is used to initialise a working directory containing

Terraform configuration files.

* During init, the configuration is searched for module blocks, and the source code for

referenced modules is retrieved from the locations given in their source arguments.

* Terraform must initialize the provider before it can be used.
* Initialization downloads and installs the provider’s plugin so that it can later be executed.
* Initializes the backend configuration.
* It will not create any sample files like example.tf
* Init Terraform and don’t ask any input

$ terraform init input= false

Change backend configuration during the init

$ terraform init -backend-config=cfg/s3.dev.tf - reconfigure is

used in order to tell terraform to not copy the existing state to the new remote state

location.

Terraform plan

* The terraform plan command is used to create an execution plan.
* It will not modify things in infrastructure.
* Terraform performs a refresh, unless explicitly disabled, and then determines what

actions are necessary to achieve the desired state specified in the configuration files.

* This command is a convenient way to check whether the execution plan for a set of

changes matches your expectations without making any changes to real resources or to the state.

$ terraform plan

Terraform Apply

* The terraform apply command is used to apply the changes required to reach the desired state of the configuration.
* Terraform apply will also write data to the terraform.tfstate file.

Once the application is completed, resources are immediately available.

$ terraform apply

Apply and auto approve

$ terraform apply -auto-approve

Apply and define new variables value

$ terraform apply -auto-approve -var tags-repository\_url=${GIT\_URL}

Apply only one module

$ terraform apply -target=module.s3

This -target option works with terraform plan too.

Terraform Refresh

* The terraform refresh command is used to reconcile the state Terraform knows about
* (via its state file) with the real-world infrastructure.
* This does not modify infrastructure but does modify the state file.

Terraform Destroy

* The terraform destroy command is used to destroy the Terraform-managed
* infrastructure.
* terraform destroy command is not the only command through which infrastructure can be destroyed.

$ terraform destroy

* You can remove the resource block from the configuration and run terraform apply this way you can destroy the infrastructure.
* A deletion plan can be created before:

$ terraform plan –destroy

* -target option allow to destroy only one resource, for example a S3 bucket:

$ terraform destroy -target aws\_s3\_bucket.my\_bucket

Terraform Validate

* The terraform validate command validates the configuration files in a directory.
* Validate runs checks that verify whether a configuration is syntactically valid and thus
* primarily useful for general verification of reusable modules, including the correctness of attribute names and value types.
* It can run before the terraform plan.
* Validation requires an initialized working directory with any referenced plugins and
* modules installed.

$ terraform validate

* Terraform fmt – set the alignment of configuration files, rewrites config files to canonical format

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" {

ami\_id=””

# ...

provisioner "local-exec" {

command = touch a

}

}

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" {

# ...

}

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 Format

The terraform fmt

* command is used to rewrite Terraform configuration files to a canonical format and style.
* where all configurations written by team members needs to have a proper

style of code

Terraform Taint….deprecated

* The terraform taint command manually marks a Terraform-managed resource as

tainted, forcing it to be destroyed and recreated on the next apply.

* This command will not modify infrastructure but does modify the state to mark a

resource as tainted.

* Once a resource is marked as tainted, the next plan will show that the resource will be

destroyed and recreated, and the next application will implement this change.

* For multiple sub-modules, the following syntax-based example can be used

Terraform taint resorcetype.logicalname

Terraform Untaint

The terraform untaint command manually unmark a Terraform-managed resource as

tainted, restoring it as the primary instance in state.

Terraform replace

This is the command replacement of terraform taint

Terraform plan/apply -replace resource type.logicalname

Terraform Import

* Terraform can import existing infrastructure.
* This allows you to take resources that you’ve created by some other means and bring

them under Terraform management.

* The current implementation of Terraform import can only import resources into the

state.

* It does not generate a configuration. Because of this, prior to running terraform import, it is necessary to write a resource configuration block manually for the resource, to which the imported object will be mapped.

terraform import resourcename.logicalname id

terraform import aws\_instance.myec2 instance-id

Terraform Show

* The terraform show command is used to provide human-readable output from a state file.
* terraform show -json will show a JSON representation of the plan, configuration, and

current state.

Terraform plan -destroy

The behaviour of any terraform destroy command can be previewed at any time with an

equivalent terraform plan -destroy command.

Variables

Variables play an important part in Terraform configuration when you want to manage

infrastructure.

Variable Types

* Strings, Numbers, Boolean, List, or Maps. We can define a default value.
* The type argument in a variable block allows you to restrict the type of value that will be accepted as the value of a variable.

variable "vpcname" {

type = string

default = "myvpc"

}

List: List is the same as array. We can store multiple values Remember the first value is

the 0 position. For example, to access the 0 position is var.mylist[0]

variable "mylist" {

type = list(string)

default = ["Value1", "Value2"]

}

Map: Map is a Key-Value pair. Key is needed to access the value.

variable "ami\_ids" {

type = map

default = {

"london" = "image-abc"

"ireland" = "image-def"

"paris" = "image-xyz"

}

}

use var.ami\_ids["paris"] to fetch the corresponding value.

Structural Data Types

* A structural type allows multiple values of several distinct types to be grouped together as a single value.
* List contains multiple values of the same type while objects can contain multiple values of different types.

Input Variables

Input variables serve as parameters for a Terraform module, allowing aspects of the

module to be customized without altering the module’s own source code, and allowing

modules to be shared between different configurations.

variable "image\_id" {

type = string

}

Input variables are created by a variable block, but you reference them as attributes on

an object named var.

resource "aws\_instance" "example" {

instance\_type = "t2.micro"

ami = var.image\_id

}

Because the input variables of a module are part of its user interface, you can briefly

describe the purpose of each variable using the optional description argument:

variable "image\_id" {

type = string

description = "The id of the machine image (AMI) to use for the server."

}

The description should concisely explain the purpose of the variable and what kind of

value is expected.

Assigning Values to Input Variables

When variables are declared in a module or configurations, they can be set in a number

of ways.

1. Manually set a variable when we run Terraform plan

If values are set, then it will ask at runtime.

2. CLI Variables

To specify individual variables on the command line, use the -var option when running

the terraform plan and terraform apply commands:

terraform apply -var="image\_id=ami-abc123"

terraform apply -var='image\_id\_list=["ami-abc123","amidef456"]'

terraform apply -var='image\_id\_map={"us-east-1":"amiabc123","us-east-2":"ami-def456"}'

3. TFVARS files

To set lots of variables, it is more convenient to specify their values in a variable

definitions file (with a filename ending in either .tfvars or .tfvars.json) and then specify

that file on the command line with -var-file:

terraform apply -var-file="testing.tfvars"

4. Auto tfvars files

Terraform also automatically loads a number of variable definitions files if they are

present:

• Files named exactly terraform.tfvars or terraform.tfvars.json.

• Any files with names ending in .auto.tfvars or .auto.tfvars.json.

5. Environment Variables

As a fallback for the other ways of defining variables, terraform searches the

environment of its own process for environment variables named TF\_VAR\_ followed by

the name of a declared variable.

$ export TF\_VAR\_image\_id=ami-abc123

$ terraform plan

Variable Definition Precedence

Terraform loads variables in the following order, with later sources taking precedence

over earlier ones:

• Environment variables

• The terraform.tfvars file, if present.

• The terraform.tfvars.json file, if present.

• Any \*.auto.tfvars or \*.auto.tfvars.json files, processed in lexical order of their

filenames.

• Any -var and -var-file options on the command line, in the order they are

provided.

* If the same variable is assigned multiple values, terraform uses the last value it finds.

Output Values

The terraform output command is used to extract the value of an output variable from

the state file.

resource "aws\_instance" "myec2" {

ami = var.image\_id

instance\_type = "t2.micro"

}

output "instance\_id" {

value = aws\_instance.myec2.id

}

If we run a apply we can see the next message:

Local Values

* A local value assigns a name to an expression, allowing it to be used multiple times

within a module without repeating it.

* The expression of a local value can refer to other locals, but as usual reference cycles are not allowed. That is, a local cannot refer to itself or to a variable that refers (directly or indirectly) back to it.
* It’s recommended to group together logically related local values into a single block,

particularly if they depend on each other.

locals {

# Ids for multiple sets of EC2 instances, merged together

instance\_ids = concat(aws\_instance.blue.\*.id, aws\_instance.green.\*.id)

}

Data Source

Data sources allow data to be fetched or computed for use elsewhere in Terraform configuration.

data "aws\_ami" "example" {

most\_recent = true

owners = ["self"]

tags = {

Name = "app-server"

Tested = "true"

}

}

Reads from a specific data source (aws\_ami) and exports results under “app\_ami”

Declarative & Imperative

* Declarative- shows what you want to do(You declare to your IaC tool what you want, but not how to get there.) ex: terraform
* Imperative-How you want to do ex: c c++

Diff b/w yaml,json,xml

* XML supports complex data types such as charts, images, and other non-primitive data types.
* JSON supports only strings, numbers, arrays, Boolean, and objects.
* YAML, supports complex data types such as date and time stamps, sequences, nested and recursive values, and primitive data types.
* HCL is a syntax and API specifically designed for building structured configuration formats.

difference among provisioning deployment and orchestration

provisioning : Prepare resources for use ex: ansible

deployment: Install software on resources ex: terraform,k8s

orchestration: co-ordination and managing the resources ex: k8s

mutable and immutable infrastructure

mutable: Infrastructure that can be modified after it is deployed

immutable: Infrastructure that cannot be modified after it is deployed

terraform both if you change ec2 ami immutable, for tags mutable

Dependencies

* Explicitly specifying a dependency is only necessary when a resource relies on some

other resource’s behaviour but doesn’t access any of that resource’s data in its

arguments.

resource "aws\_instance" "example" {

ami = "ami-a1b2c3d4"

instance\_type = "t2.micro"

depends\_on = [aws\_iam\_role\_policy.example]

}

Workspace

* Terraform starts with a single workspace named “default”.
* The workspace feature of Terraform allows users to switch between multiple instances of a single configuration with a unique state file.
* For local states, terraform stores the workspace states in a directory called terraform.tfstate.d.

Workspace commands

1. The terraform workspace new command is used to create a new workspace

and switched to a new workspace.

2. The terraform workspace list command is used to list all existing workspaces.

3. The terraform workspace select command is used to choose a different

workspace to use for further operations.

4. The terraform workspace delete command is used to delete an existing

workspace.

5. The terraform workspace show command is used to output the current

workspace.

Note: Terraform Cloud and Terraform CLI both have features called “workspaces,” but

they’re slightly different.

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=false flag as well as the -target flag to work

around this. In these scenarios, the cached state is treated as the record of truth.

State Management

State Locking

When you are creating dynamodb table partition key should be LOCK\_ID

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

Standard Backend Types

AWS S3 bucket.

AWS S3 is typically the best bet as a remote backend for the following reason

\* It’s a managed service, so no need to manage infrastructure.

\* It supports encryption at rest.

\* It support locking via DynamoDB

\* It supports versioning, so you can roll back to an older version.

terraform {

backend "s3" {

bucket = "mybucket"

key = "path/to/my/key"

region = "us-east-1" dynamodb\_table = "terraform-locks"

encrypt = true

}

}

Terraform will automatically detect that you already have a state file locally and prompt

you to copy it to the new S3 backend. If you type in “yes,” you should see:

Successfully configured the backend "s3"! Terraform will automatically use this backend

unless the backend configuration changes.

After running this command, your Terraform state will be stored in the S3 bucket.

Note: GitHub is not supported as backend type

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.

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 Modules

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

}

Call to the module example:

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

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

}

Terraform Cloud

Terraform Cloud (TFC) is a free to use, self-service SaaS platform that extends the

capabilities of the open source Terraform CLI and adds collaboration and automation

features.

The remote backend stores Terraform state and may be used to run operations in

Terraform Cloud. When using full remote operations, operations like terraform

plan or terraform apply can be executed in Terraform Cloud’s run environment, with log

output streaming to the local terminal.

Sentinel is an embedded policy-as-code framework integrated with the HashiCorp

Enterprise products.

Sentinel is a proactive service.

Note: Terraform Cloud always encrypts the state at rest and protects it with TLS in transit.

Terraform Enterprise

Terraform Enterprise provides several added advantages compared to Terraform Cloud.

Some of these include:

• Single Sign-On

• Auditing(inspection)

• Private Data Centre Networking

• Clustering

###################################################################

1. Zeal Vora course in Udemy.

2. Bryan Krausen practice sets in Udemy.

3. Andrew Brown YouTube course for quick revision.

4. Lots of hands-on Labs: Kalyan Reddy Daida + Derek Morgan course in Udemy.