TERRAFORM

IaC: Infrastructure as Code. Write and execute code to define, deploy, update, and destroy your infrastructure.

IaC Categories:

- Ad hoc scripts: Manual scripts which developers write in their favorite language to perform an ad-hoc (one time) task.

- Configuration Management Tools: Ex: Chef, Ansible, Puppet etc. Provides the ability to install and manage software on existing resources. Declarative in nature (How you want?)

- Server Templating tools: Ex: Docker, Packer, Vagrant. Instead of launching server(s), create an image of the server (snapshot) which is self-contained and can be deployed. Guarantees that the image works the same in all environments (dev, staging and prod).

- Orchestration tools: Ex: Kubernetes, ECS, Docker Swarm etc. Used for managing the server templating tools.

- Provisioning tools: Ex: Terraform, CloudFormation, Resource Manager, Cloud Deployment. Allows for creation of resources. Declarative in nature (What you want?)

Benefits of IaC:

- Self-service
- Speed and safety
- Documentation and Version control
- Validation
- Reuse
- Idempotence: Running the code which produces same result when ran several times is called Idempotence

Flow: Terraform configuration file -> Provider -> Cloud

Provider: Cloud provider - AWS, Google, Azure etc. A provider is responsible for understanding API interactions and exposing resources.

```
provider "aws" {
        region     = "us-west-2"
        access_key = "my-access-key" # Use only for development
        secret_key = "my-secret-key" # Use only for development

        # assumes role to perform action instead of access and secret access keys
        assume_role {
```

```
                <details of role to assume>
        }
}
```

- Another way is to use aws config

- "alias" will allow to create two same provider configurations

Example:

```
provider "aws" {
        region = "ap-southeast-1"
}
provider "aws" {
        region = "ap-southeast-2"
        alias = "sydney"
        Profile = "<profile name>" # here profile name refers to the AWS profile
        created using aws cli
}

resource "aws_instance" "MyEC1" {
        instance_type = "t2.micro"
        provider = "aws.sydney"
}
```

Provider version notations:

>=1.0: Greater than equal to version 1.0
<=1.0: Less than equal to version 1.0
~>2.0: Any version in 2.x range
>=2.0,<=2.30: Any version between 2.0 and 2.30

Configuration files: Declarative representation of the infra. Should end with .tf or .tf.json

Resources:

- Describes one or more infrastructure objects such as VM, LB etc

```
        resource "aws_instance" "web" {
                ami         = "ami-a1b2c3d4"
                instance_type = "t2.micro"
        }
```

Important arguments:

Depends on: To handle hidden resource dependencies that Terraform can't automatically infer.
Example: https://www.terraform.io/docs/configuration/resources.html#depends_on-explicit-resource-dependencies

Lifecycle: Affects how terraform manages the infrastructure. Available options:
- create_before_destroy
- prevent_destroy
- ignore_changes
https://www.terraform.io/docs/configuration/resources.html#lifecycle-lifecycle-customizations

terraform init

- Downloads and initializes the provider
- Should be used to initialize remote backend to store state files
-To upgrade to latest acceptable version of provider: terraform init -upgrade

terraform plan

- Would display the changes to be made to the infra. Performs terraform refresh automatically.
- Can save the generated terraform plan to specific path
- Once saved the plan can then be used with terraform apply to be certain that only the saved plan can be applied

terraform plan -out=path

Example:
terraform plan -out=examplefile
terraform apply examplefile

terraform refresh

Fetches the current state of the infrastructure. Updates state files based on infrastructure implementation

terraform apply

- Applies the terraform code and creates resources in infrastructure
- Can concurrently apply the changes – default to 10 concurrent operations. Change default using –parallelism flag
- Apply command will ask user for approval before applying the changes. For auto approval, set the flag –auto-approve [terraform apply –auto-approve]
- If the state file is already saved to external file, then use –state=path flag to set the path to state file
- Can target the apply only to certain resource [terraform apply –target=aws_instance.MYEC2]
- To restrict asking user to input for variables if default value is not specified for variable use –input=false [Environment variable alternative – export TF_INPUT= "false" or export TF_INPUT=0]

Terraform destroy:

terraform destroy: Destroys all resources

terraform destroy -auto-approve: Will not ask for confirmation from the user before deleting

terraform destroy -target aws_instance.MyEC2: To delete only the specific resource

terraform show

Displays the current terraform state (tfstate file info)

terraform fmt

To format the source code files according to terraform standards

terraform console

Opens up terraform console for trying out terraform functions

terraform validate

Checks whether the configuration file is syntactically valid

Desired state vs Current state:

Desired state: Infrastructure configuration defined in terraform

Current state: What exactly is currently running in infrastructure

Terraform will plan to match the desired state to the current state. If there is any difference between both, the desired state will take the preference.

Variable assignments:

Four ways we can pass variable value:

Variable defaults:

```
variable “instance_type” {
        default = “t2.micro”
}
```

Command line flags:

terraform apply -var=”instance_type=t2.small”

From a file:

- Create a file terraform.tfvars (by default terraform reads variable values from this file if it exists. Recommended for production use)

instance_type = “t2.large”

- If we want to create custom file to hold variable values ex: custom.tfvars, then we need to explicitly tell terraform to use the custom file during terraform plan/apply

terraform plan -var-file="custom.tfvars"

Environment variables: (below examples for linux)
export TF_VAR_<variable name> = "<value>"
export TF_VAR_instnace_type = "t2.large"

If no default value specified, terraform apply will prompt user to keyin the value at run time

Data Types:

- **string**: Example: "t2.micro"

```
variable "instance_type" {
        type = string
        default = "t2.micro"
}
Usage: resource "aws_instance" "MyEC2" { instance_type = var.instance_type}
```

*** older way (before 0.12 verison) of referencing variable value: ${var.instance_type} but still being used in certain circumstances**

- **number**: Example: 8080

```
variable "port_number" {
        type = number
        default = 8080
}
Usage: resource "aws_security_group" "MySG" { ingress
{ to_port = var.port_number}}
```

- **bool**: Example: true/false

```
variable "isDefault" {
        type = bool
        default = true
}
Usage: resource "aws_vpc" "defaultVPC" { detault = var.isDefault }
```

- **list**: Example: ["ap-southeast-1","ap-southeast-2"]

```
variable "list_regions" {
        type = list(string)
        default = ["ap-southeast-1","ap-southeast-2"]
}
Usage: resource "aws_instance" "MyEC2" { instance_type = var.list_regions["ap-
southeast-1"]}
```

- **map**: Key value pairs.

```
variable "instance_type_in_regions" {
```

```
        type = map
        default = {
                us-east-1 = “t2.micro”
                us-east-2 = “t2.micro”
                us-east-3 = “t2.micro”
        }
}
Usage: resource “aws_instance” “MyEC2”
{ instance_type = var.instance_type_in_regions [“us-east-1”]}
```

- **set**: Similar to list. List can have duplicate data. Set cannot have duplicate data.

```
variable “list_regions” {
        type = set(string)
        default = [“ap-southeast-1”,”ap-southeast-2”]
}
Usage: resource “aws_instance” “MyEC2” { instance_type = var.list_regions[“ap-
southeast-1”]}
```

- **object**:

```
variable “docker_config” {
        type = list(object({
                        internal = number
                        external = number
                        protocol = string
                })))
        default = [{
                internal = 8080
                external = 80
                protocol = “HTTP”
        }]
}
```

- **tuple**: Similar to list, but can have different data types in it

```
variable “db_params” {
        type = tuple([string, number, bool])
        default = [“ap-southeast-1”, 3306, true]
}
```

Count: Parameter: Allows to scale the resource based on the counter parameter specified.

```
resource “aws_iam_user” “lbuser” {
        name = “loadbalancer”
        path = “/system”
        count = 3 # Creates 3 users with same name
}
```

Count Index:

```
resource “aws_iam_user” “lbuser” {
        name = “loadbalancer.${count.index}”  # Assigns a different name based
        on count index
        path = “/system”
        count = 3 # Creates 3 users with same name
}
```

Can also work with list variable:

```
variable “elb_names” {
        type = list
        default = [“dev-lb”, “stage-lb”, “prod-lb”]
 }
 resource “aws_iam_user” “lbuser” {
        name = var.elb_names[count.index]  # Picks a name from list based on
        count index
        path = “/system”
        count = 3
}
```

Conditional Expression (Ternarary operator):

- Shortcut for if-else condition.

```
variable “isDev” {
        default = true
}
resource “aws_instance” “MyEC2Dev” {
        ami = “test”
        instance_type = “t2.micro”
        count = var.isDev == true ? 2 : 0 # Create production instances only
        if isDev is true
}
resource “aws_instance” “MyEC2Prod” {
        ami = “test”
        instance_type = “t2.large”
        count = var.isDev == false ? 2 : 0  # Create production instances only
        if isDev is false
}
```

Local Values:

A local value assign a name to an expression which could be used multiple times ***in a file*** without repeating it.

```
locals {
        common_tags = {
                Name = “Dev Instance”
                Team = “SharePoint”
        }
}
resource “aws_instance” “MyEC2Dev” {
        ami = “test”
        instance_type = “t2.micro”
        tags = local.common_tags
}
resource “aws_instance” “MyEC2Dev1” {
        ami = “test”
        instance_type = “t2.micro”
        tags = local.common_tags
}
```

Functions:
https://www.terraform.io/docs/configuration/functions.html

Data Sources:

Retrieves data from provider to be used in terraform configuration. Example: AMI id in different regions is different / Retrieve subnet id’s from default VPC

```
 # get default subnet
data "aws_vpc" "default" {
   default = true
}

# get subnet id’s from default VPC
data "aws_subnet_ids" "default" {
   vpc_id = data.aws_vpc.default.id
}
```

Debugging Terraform:

- Detailed logs can be enabled by setting TF_LOG environment variable.
- Can be set to one of the log levels – TRACE, DEBUG, INFO, WARN or ERROR
- Logs can be exported to files by setting TF_LOG_PATH. (export TF_LOG_PATH=/tmp/tflogs.log)

Loading of files in the folder:

- If there are 4 files – web.tf, app.tf, sg.tf etc, terraform will load all the configuration files within the directory in alphabetical order
- Instead of writing all the configuration in one file, we can split into multiple files

Dynamic Blocks:

- There could be cases where we need to add repetable nested blocks of code which when implemented leads to repeatable and long code which is unmanageable

Example AWS Security Group implementation which can have multiple ingress rules:

```
resource “aws_security_group” “SG” {
        ingress {
                from_port = 8080
                to_port = 8080
                protocol = “HTTP”
                cidr_blocks = [“0.0.0.0/0”]
        }
         ingress {
                from_port = 8081
                to_port = 8081
                protocol = “HTTP”
                cidr_blocks = [“0.0.0.0/0”]
        }
}
```

- Dynamic Blocks can helps to dynamically construct repeatable code ex: “ingress” rule above.
- Can be provided inside resource, data, provider and provisioner blocks

Example with dynamic block:

```
variable ingress_ports {
        type = list(number)
        default = [8080,8081]
}
dynamic “ingress” {
        for_each = var.ingress_ports
        iterator = port  # optional argument, if ommitted then use the label of the dynamic block (ingress). Otherwise use the iterator name in below code instead of “ingress” (ex: port.value)
        content {
                from_port = ingress.value
                to_port = ingress.value
                protocol = “HTTP”
                cidr_blocks = [“0.0.0.0/0”]
        }
}
```

Tainting Resources:

- Allows terraform-managed resource as tainted, forcing it to be destroyed and recreated on next apply.
- Updates the state file (tfstate file) for the resource to “tainted”
  - Will not modify the infrastructure
  - Will affect the resources that depend on the tainted resource

```
terraform taint <resource name>
```

Example:

```
resource “aws_instance” “MyEC2” {
        ami = “test”
        instance_type = “t2.micro”
}
```

```
terraform taint aws_instance.MyEC2
terraform apply (to destroy existing resource and create new resource)
```

Splat Expressions and Ouptut values:

- A splat expression provides a more concise way to express a common operation that could otherwise be performed with a for expression.

```
resource “aws_instance” “MyEC2Dev” {
        ami = “test”
        instance_type = “t2.micro”
        count = 3
}

# output value to output value of resource
output “arns”{
        # [*] will get the arn’s for all 3 instances created
        value = aws_instance. MyEC2Dev[*].arn

        description = "description of the output variable"

        # By default false. True will prevent field's value to be displayed
        during output.However, the state file still shown un-encrypted value
        sensitive = true
 }
```

Terraform Graph:

- Provides graphical represenation of the configuration.
- Output is in DOT format which can be converted to image
- DOT file can be visualized using Graphviz tool

```
terraform graph > graph.dot
```

Provisioners:

Provisioners are used to execute scrips on a local or remote machine as part of resource creation or destruction.

```
resource “aws_instance” “MyEC2Dev” {
        ami = “test”
        instance_type = “t2.micro”

        provisioner “remote-exec” {
                # What scripts to run during startup
                inline {
                        “sudo amazon-linux-extras install -y nginx1.12”,
                        “sudo systemctl start nginx”
                }

                # How to make connection to the VM instance
                connection {
                        type = “ssh”
                        host = self.public_ip
                        user = “ec2-user”
                        private_key = “${file(“./terraform.pem”)}”
                }
        }
}
```

Types:

1. **local-exec**: Invokes local executable after resource is created. No “connection” block is required as the provisioner executes the command locally
2. **remote-exec**: Invokes a script on a remote resource after it is created. This can be used to run a configuration management tool like Ansible, Puppet, Chef etc. Supports both ssh and winrm connections

Modules:

- Provides reusable code. Avoids repetitive code
- External modules can be referenced using "module" block

Example: EC2.tf

```
resource “aws_instance” “MyEC2Dev” {
        ami = “test”
        instance_type = “t2.micro”
}
```

Main.tf

```
module "myec2" {
```

```
        Source = "./EC2"
}
```

In the above example, since "t2.micro" is hardcoded, the code importing the module cannot update the instance_type.

If the instance_type needs to be overwritten in the calling code,
the instance_type defined in module should be getting its value from a variable.

Example:

```
variable "instance_type"{
        default = "t2.micro"
}
resource “aws_instance” “MyEC2Dev” {
        ami = “test”
        instance_type = var.instance_type
}
```

With the above example, the calling code can override the instance_type value.

Main.tf

```
module "myec2" {
        Source = "./EC2"
        Instance_type = "t2.large"
}
```

Terraform Registry:

Pre-built modules that allows for quickly deploying common infrastructure configurations.
https://registry.terraform.io/

```
module "security-group" {
         source  = "terraform-aws-modules/security-group/aws"
         version = "3.12.0"
         # insert the 2 required variables here
}
```

The above module can be used in our existing terraform configuration to create security groups.

Workspaces:

Allows for separation of environments.

Commands:

terraform workspace show: Show current workspace

terraform workspace list: To list all existing workspaces
terraform workspace select <workspace name>: To switch to the workspace specified
terraform workspace delete <workspace name>: To delete the workspace specified
terraform workspace new <workspace name>: To create new workspace specified

- In code, use terraform.workspace to retrieve the current workspace name
- Terraform.tfstate.d: Folder created by terraform to maintain **custom** workspaces. Each custom workspace (folder) contains terraform.tfstate file when terraform apply is ran.
- The terraform.tfstate file in root folder applies to "default" workspace

State Management:

Git:

- For team collaboration, state file can be stored in VCS like Git
- Problem: Sensitive info is stored in terraform.tfstate file which should be avoided

Remote Backend:

Two types of backend:
- Standard Backend Type: Used for state storage and locking
- Enhanced Backend Type: Standard Backend Features + Remote Management

Example:

```
terraform{
        backend "s3" {
                bucket = "bucket name"
                key = "key"
                region = "us-east-1
        }
}
```

State Locking:

- During performing any action, terraform would lock the state file
- When the state file is stored in remote backend, state locking is not available
- For S3, we can use DynamoDB (NoSQL db) for record locking

Terraform commands for state management:

terraform state list: Lists all resources from state file
terraform state mv: Moves item with terraform state. Useful when renaming an existing resource without destroying and recreating it. Takes a backup of state prior to saving any changes.
terraform state pull: Manually download and output the state from remote state

terraform state push: Manually upload a local state file to remote state
terraform state rm: Remove items from terraform state. Items removed from terraform state are not monitored. Resources created in the infra are not removed
terraform state show: Show the attributes of a single resource in the state

<to be updated>

Terraform Import:

Links manually created resource with the manually created terraform configuration
Example: terraform import aws_instance.EC2 <instance id taken from AWS>

Terraform Cloud:

<to be updated>