Terraform: Samson Rapando

Use this document to take notes and track your learning journey week by week.

Course: Terraform for the Absolute Beginners with Labs

Week Start Date	Week End Date	Topics/ Tasks	Links to resources used.
Aug 10, 2025	Aug 16, 2025	Introduction to Terraform	Topics 1,2
Aug 17, 2025	Aug 23, 2025	Basic Terraform, Setting & Using Variables, Resource Attributes and Output Variables State	Topics 3,4,5 Registry AWS CLI DOCS AWS CLI GUIDE
Aug 24, 2025	Aug 30, 2025		
Aug 31, 2025	Sep 6, 2025		
Sep 7, 2025	Sep 13, 2025		
Sep 14, 2025	Sep 20, 2025		
Sep 28, 2025	Oct 4, 2025	Workshop & Demo [We'll fill in the topic here]	

Introduction and Basics.

Terraform

Language: HCL (Hashicorp Configuration Language)

File Extensions: *.tf

Resource in terraform is anything managed by terraform e.g a file, a vm, database.

Providers are kind of middleware that allow you to use terraform to create and manage resources.

HCL Basics

A HCL file can have blocks and arguments. A block is defined within curly braces and arguments are in pairs.

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

For example, to create a file in a directory, we can do: ./terraform-local-file/local.tf

---

A simple terraform workflow consists of 4 steps:

1. Write the config file.
2. Run `terraform init` command.
3. Run `terraform plan` command to review the execution plan
4. When ready, run `terraform apply` command.

When the tf file is updated, the resource is deleted and recreated (immutable)

If you want to destroy resources, run `terraform destroy`

Providers

Providers are downloaded (as plugins) when `terraform init` is run.
They can be found on the Terraform Registry

There are 3 tiers:

1. Provided and maintained by Hashicorp e.g. AWS, GCP, Local
2. Partner provider: owned and maintained by 3rd party which coordinate with Hashicorp e.g Digital Ocean, Heroku
3. Community providers: Individual contributors.

`terraform init` can be run any number of times.
Plugins are installed in `./terraform` directory.

The plugin name format is: `provider/type` e.g `hashicorp/local`

A plugin can be prefixed by the provider registry, if not set, it defaults to `registry.terraform.io`

By default, the latest version is installed. In case this might make breaking changes, specify the version.

Configuration Directory.

You can have multiple .tf files. A single tf file can also have multiple config blocks e.g. `main.tf`

```
resource "local_file" "pet" {
    filename = "./pet.txt"
    content = "We love Pets!"
}

resource "local_file" "cat" {
    filename = "./cat.txt"
    content = "My favorite pet is Mr. Whiskers"
}
```

We can also have the following config files:

File Name	Purpose
`main.tf`	Main config file containing resource definition
`variables.tf`	Contains variable declaration
`outputs.tf`	Contains outputs from resources
`provider.tf`	Contains provider definitions

Multiple Providers and Resources

E.g local file and random to generate random pet names. Code in multiple-providers

---

Variables

Variables are set in `variables.tf`

```
variable "filename" {
    default = "/root/pets.txt"
    type = string # optional: string, number, bool, any (default)
    # other types: list, map, object, tuple
    description = "the description"
}

# list
variable "prefix" {
    default = ["Mr", "Mrs", "Sir"] # index begins at 0
    type = list # or list(string)
}

# usage
resource "local_file" {
    filename = var.filename
    content = var.prefix[0]
}
```

```
19
20 # maps
21 variable file-content {
22     type = map # or map(string) : data type is of the value not key
23     default = {
24         "statement1" = "We love pets!"
25         "statement2" = "We love animals!"
26     }
27 }
28
29 # usage
30 var.file-content["statement1"]
31
32 # sets are same as lists only that the values don't repeat e.g
33 variable "prefix" {
34     type = set(string)
35     default = ["Mr", "Mrs"]
36 }
37
38 # objects : complex data structures
39 variable "bella" {
40     type = object({
41         name = string
42         color = string
43         age = number
44         food = list(string)
45         favorite = bool
46     })
47     default = {
48         name = "Bella"
49         color = "Black"
50         age = 1
51         food = ["Fish", "Chips"]
52         favorite = true
53     }
54 }
55
56 # tuple similar to list, but can have multiple data types
57 variable kitty {
58     type = tuple([string, number, bool])
59     default = ["cat", 7, true]
60 }
61 # usage:
62 var.kitty
```

to use it in `main.tf`

```
1 resource "local_file" "pet" {
2     filename = var.filename
3     content = "Using variables"
4 }
```

To make updates, you can just make changes to the Terraform | HashiCorp Developer and the main.tf won't be changed. This is important if you're setting up the same resources with different names, specs etc for dev and production.

Using variables.

When variables don't have default values, you can either:

- enter the variables during apply or
- add them to apply on CLI e.g `terraform apply -var "filename=/hello.txt" -var "prefix=mrs"`

We can also use environment variables

```
export TF_VAR_filename="/root/hello.txt"
export TF_VAR_prefix="Mr"
terraform apply
```

We can also use a environment file `terraform.tfvars` or `terraform.tfvars.json`

```
filename = "/root/hello.txt"
prefix = "Mrs"
```

Variable definition precedence: If multiple exist, the one lower on the list is used.

1. Env Variables
2. terraform.tfvars
3. *.auto.tfvars (alphabetic order)
4. -var or var-file

Resource Attributes

Linking two/ more resources. for example, when we want to use a random pet name in the pet file.

Reference the attribute that is returned from the resource (check in `terraform apply` command.

```
resource "local_file" "pet" {
  filename var.filename
  content = "My favorite pet is ${random_pet.my-pet.id}"
}

resource "random_pet" "my-pet" {
  prefix = var.prefix
  separator = var.separator
  length = var.length
```

```
}
```

Resource Dependencies

Terraform creates resources in order of dependency: `random_pet` then `local_file`
When destroying, it happens in reverse order.

We can do this manually (explicit dependency). Useful when references are not used but one resource still depends on the other working. Implicit dependency is when data is referenced in another resource.

```
resource "local_file" "pet" {
  filename var.filename
  content = "My favorite pet is Mr. Cat"
  depends_on = [
    random_pet.my-pet
  ]
}

resource "random_pet" "my-pet" {
  prefix = var.prefix
  separator = var.separator
  length = var.length
}
```

Output Variables

Used to store the values of an expression in terraform.

```
resource "local_file" "pet" {
  filename var.filename
  content = "My favorite pet is Mr. Cat"
  depends_on = [
    random_pet.my-pet
  ]
}

resource "random_pet" "my-pet" {
  prefix = var.prefix
  separator = var.separator
  length = var.length
}
```

The `random_pet` resource will provide an ID which will contain the pet name

```
resource "local_file" "pet" {
  filename var.filename
  content = "My favorite pet is Mr Cat"
  depends_on = [
    random_pet.my-pet
  ]
}

resource "random_pet" "my-pet" {
  prefix = var.prefix
  separator = var.separator
  length = var.length
}

output "pet-name" {
  value = random_pet.my-pet.id
  description = "Record the value of pet id generated by the random_pet.my-pet resource"
}
```

`terraform output` prints all outputs from the current config. To print a specific one

```
terraform output pet-name
```

Best use of output variables:

1. Printing the output on screen
2. Exporting them to external providers e.g Ansible.

For example, if you have used terraform to set up a db, the output variable could be credentials.

Terraform State

State is stored in `terraform.tfstate` file after `terraform.apply` is run. For every apply, terraform compares the current contents of resources to the stored state. If there are changes, it makes the changes and updates the state. if not, then nothing will be done because the latest state is the stored state.

Terraform relies on state to know dependency when a resource is deleted from the config file.

State also improves performance.

If `--refresh=false` is set when running terraform commands, only state is checked.

The state is useful when collaborating. In this case, put the state in a remote shared place so that every member has the same state. e.g AWS S3,

State conciderations

1. State file contains sensitive information
2. Store the state in a secure place, NOT in github.
3. Do not edit the state file unless you use `terraform state` commands.

Terraform Commands

```
# commands we've learnt so far
terraform init
terraform plan
terraform apply
terraform show
```

```
terraform validate # checks whether the syntax used is correct
terraform fmt # formats the config files.
terraform show # shows the current infrastructure as in the state of terraform.
terraform providers # show all providers
# if you want to copy the providers to another dir:
terraform providers mirror /path/tonewmirror/terraform/new_local_file
terraform output # print all output variables. Append name of variable for specific variable
terraform apply -refresh-only # does not modify the infra, but modifies the state file.
terraform graph # visual representation of configs, plan and dependencies
# to visualize the graph, use a tool e.g graphviz then
terraform graph | dot -Tsvg > graph.svg
```

Mutable vs Immutable Infrastructure

when creating a file, when we change the arguments in the resource, the file is deleted and a new one is created.

When updating OS on a server, we wouldn't want to delete the whole server.

Mutable infrastructure: the code/ software can be updated on a system. (in-place updates)

Immutable infra: Cant be updated, an update means a deletion of the old resource.

Terraform uses immutable infrastructure

If we want a resource to be created before deletion, lifecycles are used.

Lifecycle Rules

You might want to create a new resource before a new one is created/ old one should not be deleted.

```
resource "local_file" "pet" {
  filename = "./pet.txt"
  content = "I love pets"
  file_permission = "0700"

  lifecycle {
    create_before_destroy = true
  }
}
```

In case we don't want the original to be deleted

```
resource "local_file" "pet" {
  filename = "./pet.txt"
  content = "I love pets"
  file_permission = "0700"

  lifecycle {
    prevent_destroy = true
  }
}
```

ignore_changes will avoid changes depending on the attributes set.

```
resource "aws_instance" "webserver" {
  ami = var.ami
  instance_type = var.instance_type
  tags = {
    Name = "ProjectAWebserver"
  }
  lifecycle {
    ignore_changes = [tags]
    # Ignore changes that are made in tags attribute. If a change is made in tags
    # do not initiate an update the resource
    # use all to prevent any updates from changes in the resource.
  }
}
```

Data sources

read data from external places e.g. a text file. E.g

```
echo "Dogs are awesome!" > dog.txt
```

In Terraform

```
resource "local_file" "pet" {
  filename = "./pets.txt"
  content = data.local_file.dog.content
}

# local_file is the type of resource we're reading data from
data "local_file" "dog" {
  filename = "./dog.txt"
}


```

Meta Arguments

We've already used: `depends_on` and `lifecycle`

When creating multiple instances of the same resource.

Count

```
resource "local_file" "pet" {
  filename = var.filename[count.index]
  content  = "This is a sample file"
  #   count    = 3 # when set to a static value, only that number will be created
  count = length(var.filename) # this allows us to update the list in filename
}


variable "filename" {
  type    = list(string)
  default = ["./cat.txt", "./bear.txt"]
}

```

foreach

```
resource "local_file" "pet" {
  filename = each.value
  for_each = var.filename # only works with a map/set or do toset({arg})
}

# when using this, when apply is run and one file has been removed, only
# the removed resource will be destroyed.
variable "filename" {
  type    = set(string)
  default = ["./cat.txt", "./bear.txt"]
}
```

Version Contraints

Providers have versioned plugins. If we want to maintain the same version:

1. Go to the registry and show versions of the plugin.
2. Choose the version you want and click 'Use provider'. You will get a block of code that specifies the provider version.

```
terraform {
  required_providers {
    local = {
      source = "hashicorp/local"
      version = "1.4.0"
      # version = "!= 2.0.0" # do not use 2.0.0
      # version = "< 1.4.0" # use  version below this
      # version = "> 1.2.0, < 2.0.0, != 1.4.0"
      # version = "~> 1.2" # this or newer version (1.2.*)
    }
  }
}

provider "local" {
  # Configuration options
}
```

AWS

(Will use AWS for class but for demo, will use GCP/ another cloud provider)

For Demos, I will use LocalStack

```
brew install localstack/tap/localstack-cli
```

Check ~/.zshrc and ~/.aliases for configs

AWS IAM CLI REFERENCE

Create users with programmatic access and download their credentials (id and key).

Plugin : Terraform Registry

To create a user:

```
# provide block with credentials and region details to be used.
```

```
provider "aws" {
  region = "us-west-2"
  access_key = "ACCESSKEY"
  secret_key = "SECRET"
}

#### IF USING A LOCAL MOCK OF AWS (LocalStack)
provider "aws" {
  region                      = "us-east-1"
  skip_credentials_validation = true
  skip_requesting_account_id  = true

  endpoints {
    iam                       = "http://aws:4566" # replace with real url
  }
}
####

resource "aws_iam_user" "admin-user" {
  name = "lucy"
  tags = {
    Description = "Technical Team Leader"
  }
}

resource "aws_iam_policy" "adminUser" {
  name = "AdminUsers"
  policy = file("admin-policy.json")
}

resource "aws_iam_user_policy_attachment" "lucy-admin-access" {
  user = aws_iam_user.admin-user.name
  policy_arn = aws_iam_policy.adminUser.arn
}
```

To avoid hardcoding credentials would be to create a credentials file at the root of the terraform dir (or just use environment vars and store them securely) `./.aws/credential`

```
aws_access_key_id =
aws_secret_access_key =
```

Read more info on the aws plugin docs.

Remote State

It is advised to store state remotely for:

- security of contents
- collaboration for multiple users.

Configure a remote backend to store the state file.

```
# main.tf
resource "local_file" "pet" {
  filename = "/root/pets.txt"
  content = "We love pets!"
}

# terraform.tf
terraform {
  backend "s3" {
    bucket = "name-of-bucket"
    key = "finance/terraform.tfstate"
    region = "us-west-1"
    dynamodb_table = "state-locking"
  }

}
```

Terraform State Commands

Do not update the state file.

```
terraform state <command>
```

```
# list all resources
terraform state list [options] [address]
terraform state show local_file.pet
```

```
terraform state mv [options] SOURCE DESTINATION
```

```
terraform state pull # view the state saved remotely
```

```
terraform state rm ADDRESS # remove resource,
# make sure to remove the resource from the tf file too
```

Provisioners

EC2

EC2: Elastic Cloud Compute

AMI: Amazon Machine Image (each image has an id)

Instance type: depends on the type of workload.

General use, compute, memory optimised etc

EBS (Elastic Block Storage) storage for these instance types. These storages are attached to the instance.

User data can be passed e.g a script to deploy applications e.g nginx
We use ssh keys to access the instance

The provisioner for local exec is used to save info locally, e.g the public ip of the instance that was created. The remote-exec is run on the instance.

Use provisioners sparingly.

Terraform Taint

Terraform marks a resource as tainted if it fails to apply e.g because a provisioner failed.
If we want to recreate a resource we can taint it so that it will not recreated in the next apply.

untaint untaints a previous taint.

Log Levels

```
export TF_LOG=TRACE terraform apply
```

Terraform Import

Import existing infrastructure into terraform config. For example, existing servers and dbs that were created elsewhere.

```
terraform import <resource_type>.<resource_name> <attribute>
# e.g.
terraform import aws_instance.webserver-2 i-jkvsdhvbfj
```

this does not update the configs (they are in tfstate- check new version)

Terraform Modules

When dealing with large infra, the resources can be split into multiple files, but this can still be risky, with duplication and complex.

We can use a module block to import configs from another dir.

```
# root module.
module "dev-webserver" {
  source = "../aws-iinstance" # child module
}
```

In the modules dir, we specify the resource types and use variables where details might change.

in the other dirs, we can then import the module as (the variables are set in the module import)

```
module "us_payroll" {
  source = "../modules/payroll-app"
  app_region = "us-east-1"
```

```
  ami = "ami-xxx"
}
```

Terraform functions

We've used functions such as length, file etc.

Terraform provides a console for testing functions.

```
terraform console
```

It is loaded in the context of the project and we can use the console to test the output of a function.

Copy the function call and paste it in the console.

Types of functions

- Numeric: transform and manipulate function e.g max, min `max(var.num...)`, ceil, floor
- String: manipulate/ transform string data e.g. `split(",", "fd,fd,fd")`, lower, upper, title, `substr(var.ami, 16,7)`, join
- Collection : manipulates sets, maps and lists. `index(var.ami, "AMI-X")`, `keys(var.ami)` converts keys in a map to a list. `values(var.ami)` does hte same for values, `lookup(var.ami, "ca-central-1)` returns value for key. `lookup(var.ami, "key", "default-return")`
- Type conversion

Conditional Expressions

values are type sensitive

```
8 == 8
8 != 7
6 <= 7
10 >= 1
```

Logical Expressions

```
8 > 7 && 8 < 10
```

```
resource "random_password" "password-generator" {
    length = var.length < 8 ? var.length : 8 # if length < 8  then we use 8 as default.
}

output "password" {
    value = random_password.password-generator.result
```

```
}

variable length {
    type = number
    description = "The length of the password"
}
```

Terraform Workspaces.

Terraform state stores the current status of the infrastructure.

We can have the same directory to create different infra

```
terraform workspace new ProjectA
```

List

```
terraform workspace list
```

We can then create a map variable with values whose keys are the workspace names.

We then reference this as

```
ami = lookup(var.ami, terraform.workspace)
```

This is important in situations such as configuring dev and prod environments, similar in structure but with different specs.

Create a workspace

```
terraform workspace create workspace-name
```

Switch workspaces

```
terraform workspace select ProjectA
```

With multiple workspaces, states are stored in the `terraform.tfstate.d` directory. In each, there's a directory containing the tfstate for each workspace.

Yay!