Main Concepts

Terraform

Agenda

- Terraform CLI
- Providers & Provisioners
- Resources
- Variables & Outputs
- Organize your code
- Terraform State

Terraform CLI

Terraform

Terraform CLI

- Terraform CLI (Command Line Interface) is a program written in Golang
- Available for all most used platforms (Windows, Linux, Mac, ARM, …)
- Allow the execution of manual commands but more important to be run inside automation processes
- Use Terraform code (files with .tf extension) as input for each command
- Some available commands
  - terraform init
  - terraform validate
  - terraform fmt
  - terraform plan
  - terraform apply
  - terraform destroy

Providers & Provisioners

Terraform

Terraform Providers

- Terraform allows you to build infra in different clouds but is not cloud agnostic
- Terraform embraces different cloud providers to allow building infra using the same language (HCL) and same principles
- But when you have Terraform code to build an environment in Azure, if you need to create in AWS too, you need to create new code from scratch
- What allows Terraform to communicate with all cloud providers, are the Terraform Providers

Terraform Providers

- You can have Terraform Providers for cloud providers, SaaS applications, on-prem virtualization, etc.
- Every solution that you use that gives you a CLI or a well-defined API may have a Terraform Provider
- If you need a customized provider, you can build your own using Golang
- Examples
  - Cloud providers: Azure, AWS, GCP
  - DevOps Platforms: GitHub, Gitlab, Azure DevOps
  - Monitoring: NewRelic
  - VPNs: Twingate, Tailscale

Terraform Provisioners

- A Terraform Provider can be seen as a library that you use on your code and allow you to write code to use it
- But then, you need to translate that code to something that you final provider knows
- For example, if you use Azure provider to write the code to create a VM, you need to translate from that code to Azure instructions
- That's the role of the provisioners!
- Using Azure example, the provisioner receives the Terraform code and translate for **az CLI** commands to reach your Azure Resource Manager
- These provisioners need to be installed by **terraform init** code and must be always the first command to be run

How to define providers?

- You can add as many providers you want to your Terraform code
- For each provider you should always specify a version and specific properties needed by the provider
- The version should be defined on a terraform config block to centralize all needed versions configuration

```
terraform {
  required_version = ">= 1.4.0"
  required_providers {
    azurerm = "3.55.0"
  }
}

provider "azurerm" {
  subscription_id = "subscription-id"
  client_id = "principal-used-for-access"
  client_secret = "password-of-principal"
  tenant_id = "tenant-id"
  alias = "arm-1"
}
```

How to set version numbers?

- On Terraform block you should specify:
  - Terraform CLI version to be used with your code
  - Version for each provider your using
- When you define a Terraform CLI version, you may have a CLI version compliant with defined on the code
- How can version number be defined:
  - '**=**': Allows only one exact version number
  - '**!=**': Excludes an exact version number
  - '**<**', '**<=**', '**>**', '**>=**': Compares the version and allows any when comparison is true. '**>= 3.0.0**' allows '**3.0.0**', '**3.0.1**', '**3.1.0**' and '**4.0.0**'
  - ´**~>**´: Allows only the rightmost version component to increment
    - ´**~> 1.0.4**': Allows versions '**1.0.5**' and '**1.0.10**' but not '**1.1.0**'
    - '**~> 1.1**': Allows versions '**1.2**' and '**1.10**' but not '**2.0**'

Resources

Terraform

Resources

- Resources are the building blocks of Terraform code
- When you author Terraform code you're defining resources and the relationship between them
- Each resource represents an element on the target provider
- Inside each resource you define unique properties that make sense on the context of that resource

Resources

- **resource** = Top level keyword
- **type** = Type of resource. Ex: azurerm_virtual_machine.
- **name** = Arbitrary name to refer to this resource. Used internally by terraform. This field cannot be a variable.

```
resource "type" "name" {
  parameter = "foo"
  parameter2 = "bar"
  list = ["one", "two", "three"]
}
```

Resources: Example

- Resource to create AWS VM
- **aws_instance** = Type of resource available on AWS provider
- **ex** = name to the resource
- **ami** and **instance_type** = properties of the AWS VM

```
resource "aws_instance" "ex"{

  ami = "ami-c58c1dd3"
  instance_type = "t2.micro"

}
```

Resources: Dependency Mapping

- By default, Terraform tries to create all resources in parallel
- But sometimes that is not possible because you have dependency between resources
- Dependency mapping is a out-of-the-box feature of Terraform that uses the code you created to define the sequence to create the resources
- Based on your code Terraform creates a dependency graph between all resources and start to create resources in that sequence

Resources: Dependency Mapping

- In some cases, mostly due to provider implementation, you many need to add an explicit dependency between resources
- Terraform detects circular references too and gives an error (without changes any resource) when that happens

```
resource "azurerm_resource_group" "hashitraining" {
  name     = "${var.prefix}-vault-workshop"
  location = var.location
}

resource "azurerm_virtual_network" "vnet" {
  name                = "${var.prefix}-vnet"
  location            = azurerm_resource_group.hashitraining.location
  address_space       = [var.address_space]
  resource_group_name = azurerm_resource_group.hashitraining.name
}
```

Resources: How to reference?

- The reference from one resource to the other, follows this pattern

resource_type.resource_name.property_name

```
resource "azurerm_resource_group" "hashitraining" {
  name     = "${var.prefix}-vault-workshop"
  location = var.location
}

resource "azurerm_virtual_network" "vnet" {
  name                = "${var.prefix}-vnet"
  location            = azurerm_resource_group.hashitraining.location
  address_space       = [var.address_space]
  resource_group_name = azurerm_resource_group.hashitraining.name
}
```

Variables & Outputs

Terraform

Variables

- Variables allow to bring more dynamic behavior to your Terraform code
- Defined on your code work as variables and parameters
- Variables can have default settings
- If you omit the default, the user will be prompted to enter a value

Variables

- Variables must have a "**name**" property
- Variables can have a "**description**" and "**default**" properties

```
variable "prefix" {
  description = "This prefix will be included in the name of most resou
}

variable "location" {
  description = "The region where the virtual network is created."
  default     = "centralus"
}
```

How to use variables?

- You may use variables value in two ways, direct or using interpolation

- Direct mode, you use the name of the variable with ´**var.**´ as prefix

```
resource "azurerm_virtual_network" "vnet" {
  address_space       = var.address_space
  location            = var.location
  name                = var.name
  resource_group_name = var.resource_group
  dns_servers         = var.dns_servers
}
```

How to use variables?

- Using interpolation, you need to put your variable inside '${ … }' and this code needs to be inside double quotes

```
resource "azurerm_virtual_network" "vnet" {
  address_space       = var.address_space
  location            = var.location
  name                = "${var.name}-vnet"
  resource_group_name = "${var.resource_group}-rg"
  dns_servers         = var.dns_servers
}
```

How to set variables value?

- You can set variables values following the next precedence list

1. Command line flag
2. Configuration file
3. Environment variable
4. Default Config
5. User manual entry - if not specified, prompt the user for entry

Outputs

- With outputs you can show or pass data that you gather after resources creation
- These values can be used an inputs for another resources to be used as properties values
- The outputs are automatically shown as log on the standard output

```
output "Vault_Server_URL" {
  value = "http://${azurerm_public_ip.vault-pip.fqdn}:8200"
}

output "MySQL_Server_FQDN" {
  value = azurerm_mysql_server.mysql.fqdn
}

output "catapp_url" {
  value = "http://${azurerm_public_ip.catapp-pip.fqdn}"
}
```

Outputs: Sensitive information

- You may protect your outputs to not be automatically write on the standard output marking as sensitive data
- The providers already do that on their resources so if you want to output a password/token/etc. you will see a redacted string
- To mark a output as sensitive, add the property '**sensitive = true**' on output definition

```
output "name" {
  description = "The name of the newly created vNet"
  value       = azurerm_virtual_network.vnet.name
  sensitive   = true
}
```

Organize your code

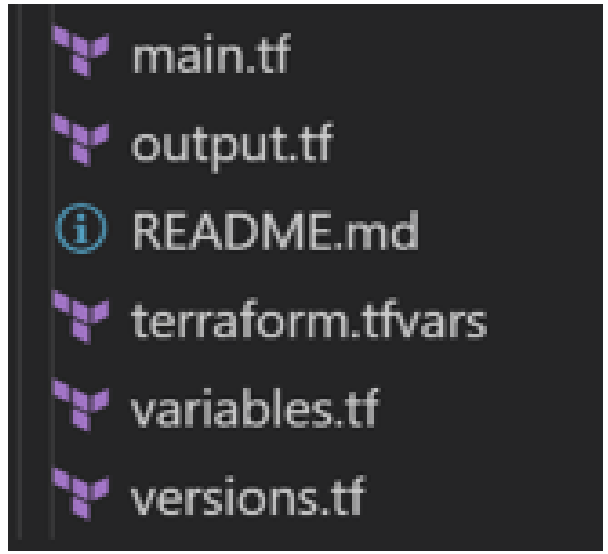
Terraform

Organize your code

- In Terraform you don’t have mandatory way to organize your code
- Terraform CLI takes all input files and internally merge all the code
- But there is a recommended way to structure your code and is followed by all community

Best practices to author code

- **main.tf**: Main structure in order to deploy the complete infrastructure directly or through calls to modules
- **output.tf**: Identify each return value of a Terraform module
- **variables.tf**: Define variables required (or not) in main.tf, we need to define the var type and it allows to you assign a default value.
- **versions.tf**: Define minimum version required for terraform and providers in a module
- **terraform.tfvars**: Assign values to variables in a file instead of arguments in a command



Terraform lock file

- After you execute terraform init command, all providers/provisioners are downloaded to .terraform folder
- Additionally, a terraform lock file (named XXXXX) is created with hash and version information about all downloaded content
- This file should be treated as source code and should be pushed to your repo to keep track of providers versions

Terraform State

Terraform

Terraform State

- **Terraform State** is a critical concept in Terraform
- Represents a **snapshot of your infrastructure** at a specific point in time
- Terraform uses this state file to **keep track of resources** you've deployed to your providers
- With it ensures that it knows the current state of your infrastructure
- Terraform State if commonly named as **tfstate**

Terraform State: Why is important?

- When you use Terraform to create resources, it needs to remember:
  - Which resources it has already created
  - The properties of those resources (IDs, IP addresses, configuration details, etc.).
  - The relationship between your resources, like which instances are attached to which networks.
- Terraform **stores this information in a state file**, usually named terraform.tfstate
- This state file is used every time you run commands like **terraform plan** or **terraform apply**
- It's used to determine the **differences between your desired configuration and the actual resources** in your cloud environment

Terraform State: Terraform State format

- Terraform state file is a JSON file that must be only managed by Terraform
- You should never update this file manually
- Terraform (using Backend Providers) have different ways to make sure the tfstate was not changed manually


```
{
  "version": 4,
  "resources": [
    {
      "type": "aws_instance",
      "name": "my_server",
      "instances": [
        {
          "attributes": {
            "id": "i-0abcdef1234567890",
            "ami": "ami-12345678",
            "instance_type": "t2.micro",
            "public_ip": "54.12.34.56"
          }
        }
      ]
    }
  ]
}
```

Terraform State: How it works?

- Initial State: When you run **terraform apply** for the first time, Terraform creates the resources and writes their current state to **terraform.tfstate**.
- State Updates: On subsequent runs, Terraform compares the existing **terraform.tfstate** with your updated configuration files to determine what changes are needed (if any).
- Tracking Changes: Terraform uses this state to efficiently plan and apply only the necessary changes, minimizing disruption to your infrastructure.

Terraform State: Plan output

- When you run terraform plan and Terraform makes this comparison between you source code, terraform state and real resources you get a list of operations to be done

```
+   create
-   destroy
-/+ replace
~   update in-place
```

Demo – Your first Terraform Code

Terraform

Lab 01 – Your first Terraform Code

Terraform