Antony Kervazo-Canut

# Terraform for Teenagers

SUITABLE FOR ADULTS

# Introduction

What is Terraform?

Terraform is an Infrastructure as Code (IaC) tool developed by HashiCorp. It allows users to define and provision cloud service infrastructure using a simple and declarative configuration language.

Why use Terraform?

Infrastructure Automation: Terraform automates the deployment and management of infrastructure, reducing the risk of human errors. State Management: Terraform maintains a state of the infrastructure, allowing for predictable modifications and deployments. Multi-cloud Support: Terraform can manage resources across multiple cloud platforms, offering great flexibility. Infrastructure as Code: Configurations are written in files that can be versioned, reused, and shared.

# Terraform Installation

```
## Linux
wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --
dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
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 install terraform

## Windows
wget -O- https://apt.releases.hashicorp.com/gpg | sudo gpg --
dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg
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 install terraform

## MacOS
brew tap hashicorp/tap
brew install hashicorp/tap/terraform
```

# Terraform Configuration

To be able to use Terraform, several prerequisites are needed in addition to Terraform itself. First, a valid SSH key:

```
# Check if an SSH key exists
ls ~/.ssh/id_rsa.pub

# If not, create one
ssh-keygen
```

For Terraform with Azure, you will also need Azure CLI on your system : Install AzureCLI
For Terraform with AWS, you will also need AWS CLI on your system :: Install AWS CLI

# Start with Terraform

```
provider "azurerm" {
  features {}
}

resource "azurerm_resource_group" "example" {
  name     = "example-rg"
  location = "France Central"
}
```

The two files mentioned here are simple. But they serve to indicate which provider you are going to use. Above: Azure, below: AWS. Due to an unfair preference, everything will be done on Azure.

```
provider "aws" {
  region = "eu-west-3" # Paris
}

resource "aws_vpc" "example" {
  cidr_block = "10.0.0.0/16"
  tags = {
    Name = "example-vpc"
  }
}
```

# terraform init

```
# Connect to Azure, importance of having AzureCLI
az login

# Initialize the configuration file and provider
terraform init
```

What does terraform init do?

- Installation of Plugins: Terraform downloads and installs the necessary plugins to interact with the specified providers in your configuration (for example, Azure, AWS).
- Initialization of the Backend: If specified, configures the backend for Terraform's state.
- Preparation of the Directory: Terraform prepares the working directory for the execution of other Terraform commands.

A success message will indicate that the initialization was successful. You are now ready to plan and apply your Terraform configurations.

# terraform plan

```
# Run terraform plan in your project directory to see what changes will be applied to your infrastructure.
terraform plan
```

What does terraform plan do?

- Configuration Analysis: Terraform analyzes your configuration files to determine the resources to create, modify, or destroy.
- Display of Changes: Terraform presents an execution plan, showing what will be changed in the infrastructure.
- Prevention of Surprises: This allows for reviewing changes before applying them, reducing the risk of unexpected errors.

The plan displays additions (+), changes (~), and deletions (-). It is crucial to read it carefully to ensure that the changes match your expectations.

```
Terraform used the selected providers to generate the following execution plan. Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.example will be created
  + resource "azurerm_resource_group" "example" {
      + id       = (known after apply)
      + location = "francecentral"
      + name     = "example-rg"
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

# terraform apply

```
# Run terraform apply to apply the changes specified in your
execution plan.
terraform apply
```

What does terraform apply do?

- Confirmation of Changes: Before applying changes, Terraform displays the plan and requests confirmation.
- Modification of the Infrastructure: If confirmed, Terraform applies the changes to the infrastructure according to your configuration.
- Update of the State: After applying the changes, Terraform updates the state file to reflect the current state of the infrastructure.

Terraform's state is crucial for maintaining alignment between your configuration and the actual infrastructure. After each application, Terraform updates its state file.

# terraform destroy

```
# Use terraform destroy to remove all resources managed by
your Terraform configuration.
terraform destroy
```

What does terraform destroy do?

- Display of Resources to Delete: Terraform displays a list of the resources it plans to delete.
- Request for Confirmation: Before proceeding, Terraform requests confirmation to ensure that you wish to delete these resources.
- Destruction of Resources: If confirmed, Terraform deletes all the listed resources, thereby cleaning up the infrastructure.

This command is useful at the end of a project, for test environments, or to completely rebuild an infrastructure. It is important to use it with caution, as it deletes all managed resources.

# State Management

Terraform uses a state file to keep track of the infrastructure and configurations. This state file is crucial for Terraform's operation.

- The Terraform state file (terraform.tfstate) records the IDs and properties of the resources created by Terraform. It is used to map real resources to your configuration and to keep track of metadata.

```
# List all resources currently managed by Terraform.
terraform state list

# Show the current state of your infrastructure as known by
Terraform.
terraform show
```

# Variables

```
project/
├── globals.tfvars
└── environments/
    └── dev/
        ├── main.tf
        ├── variables.tf
        └── provider.tf
```

When a project becomes more substantial, it's important to start architecting it. This involves beginning to separate different resources, the definition of the provider, or the variables.

For the following example, we will revisit the file for creating resources on Azure by moving the provider into provider.tf.

```
provider "azurerm" {
  features {}
}
```

# Variables

```
resource "azurerm_resource_group" "example" {
  name     = "example-resources"
  location = "France Central"
}
```

By moving the resource group into the file main.tf, we achieve this setup. However, envisioning a world with multiple environments, we will extract the variables name and location to place them in the file global.tfvars.

```
resource_group_name = "example-rg"
location = "France Central"
```

# Variables

```
variable "resource_group_name" {
  description = "The name of the resource group"
  type        = string
}

variable "location" {
  description = "The global location for the resources"
  type        = string
}

variable "name_prefix" {
  description = "Prefix for the resource group name in the
development environment"
  default     = "dev-"
  type        = string
}
```

We create the file variables.tf (mentioned above) and modify the main.tf. As our architecture is divided by environment, we prefix its name with "dev-".

```
resource "azurerm_resource_group" "example" {
  name     = "${var.name_prefix}${var.resource_group_name}"
  location = var.location
}
```

# Variables

```
# Change directory
cd environments/dev

# Since it's a different directory, we need to reinitialize
terraform init

# To successfully execute the plan, variables will be needed
terraform plan -var-file="../../globals.tfvars"

# Once you are satisfied with the outcome, you can apply
terraform apply -var-file="../../globals.tfvars"
```

```
  # azurerm_resource_group.example will be created
  + resource "azurerm_resource_group" "example" {
      + id       = (known after apply)
      + location = "francecentral"
      + name     = "dev-example-rg"
    }

Plan: 1 to add, 0 to change, 0 to destroy.
```

# Prevent Destroy

Terraform allows marking a resource to prevent its destruction through the lifecycle.prevent_destroy attribute within the resource block. However, this attribute must be used with caution, as it makes manually destroying the resource difficult without modifying the Terraform code.

```
resource "azurerm_resource_group" "example" {
  name     = var.resource_group_name
  location = var.location

  lifecycle {
    prevent_destroy = true
  }
}
```

However, this action will create an error during a destroy operation. Therefore, you will need to rethink the architecture of your Terraform project to group permanent resources together and place temporary resources on the other side.

# Modules

```
project/
├── provider.tf
├── main.tf
└── modules/
    └── azure_rg_module/
        ├── main.tf
        ├── output.tf
        └── variables.tf
```

Modules allow for the reuse of code snippets in different projects or parts of a project.

Organization: They help organize and structure Terraform code in a more readable and maintainable way.

Encapsulation: Modules can encapsulate complex logic, exposing only the necessary parameters as inputs and outputs.

# Modules
## azure_rg_module

```hcl
# main.tf
resource "azurerm_resource_group" "example" {
  name     = var.resource_group_name
  location = var.location
}
```

```hcl
# output.tf
output "resource_group_id" {
  value = azurerm_resource_group.example.id
  description = "The ID of the created resource group"
}
```

```hcl
# variables.tf
variable "resource_group_name" {
  description = "The global name of the resource group"
  type        = string
}

variable "location" {
  description = "The global location for the resources"
  type        = string
}
```

# Modules

At the root, in the main.tf file, we declare the module that we created by passing the variables directly as parameters.

The output allows logging the creation of the resource group.

```
module "azure_rg_module_first" {
  source              = "./modules/azure_rg_module"
  resource_group_name = "exempe-rg"
  location            = "France Central"
}

# To display the output
output "premier_groupe_resource_id" {
  value = module.azure_rg_module_first.resource_group_id
}
```

```
# Initialize to add the module
terraform init

# Check the plan and execute
terraform plan
terraform apply
```

# Conditions


**Terraform allows the use of logical conditions in configurations by using a combination of variables and the ternary operator. This enables dynamic adjustment of the configuration based on certain conditions.**

```
variable "create_resource" {
  description = "A flag to create or not a resource"
  type        = bool
  default     = false
}

resource "azurerm_resource_group" "example" {
  count    = var.create_resource ? 1 : 0  # Use of a
condition
  name     = "example-resource-group"
  location = "West Europe"
}
```

# Loop for

Create a resource, for example, a network security group, and use a for expression to generate tags.

```
variable "instance_names" {
  type    = list(string)
  default = ["instance1", "instance2", "instance3"]
}

resource "azurerm_network_security_group" "example" {
  name                = "nsg-example"
  location            = "West Europe"
  resource_group_name = azurerm_resource_group.example.name

  tags = { for name in var.instance_names : name => "${name}-
security" }
}
```

In this example, for each instance name in var.instance_names, a tag is created where the key is the instance name and the value is a combination of the instance name and a suffix (for example, "instance1" becomes "instance1-security").

# Loop for_each

for_each is used to iterate over each element of a map or a list, creating an instance of the resource for each element.

```
variable "instance_tags" {
  type = map(string)
  default = {
    "instance1" = "tag1"
    "instance2" = "tag2"
  }
}

resource "azurerm_virtual_machine" "example" {
  for_each            = var.instance_tags
  name                = each.key
  tags                = {"Name" = each.value}
  location            = "West Europe"
  resource_group_name = azurerm_resource_group.example.name
  # Others configurations ...
}
```

Here, for_each creates a VM for each element in var.instance_tags, using the key for the name and the value for the tag.

# Dynamic block

**Consider the creation of multiple rules in a network security policy, where the number of rules is variable.**

```
variable "nsg_rules" {
  description = "List of security rules to apply to the NSG."
  type = list(object({
    name                       = string
    priority                   = number
    direction                  = string
    access                     = string
    protocol                   = string
    source_port_range          = string
    destination_port_range     = string
    source_address_prefix      = string
    destination_address_prefix = string
  }))
  default = [
    {
      name                       = "allow-http"
      priority                   = 200
      direction                  = "Inbound"
      access                     = "Allow"
      protocol                   = "Tcp"
      source_port_range          = "*"
      destination_port_range     = "80"
      source_address_prefix      = "*"
      destination_address_prefix = "*"
    },
    {
      name                       = "deny-outbound"
      priority                   = 300
      direction                  = "Outbound"
      access                     = "Deny"
      protocol                   = "*"
      source_port_range          = "*"
      destination_port_range     = "*"
      source_address_prefix      = "*"
      destination_address_prefix = "*"
    }
  ]
}
```

# Dynamic block

In this example, nsg_rules is a variable that contains a list of objects, each object representing a specific network security rule for the Azure security group using dynamic blocks in Terraform, thereby providing a flexible and powerful method for managing network security configurations in Azure.

```
resource "azurerm_network_security_group" "example" {
  name                = "example-nsg"
  location            = var.location
  resource_group_name = azurerm_resource_group.example.name

  dynamic "security_rule" {
    for_each = var.nsg_rules

    content {
      name                       = security_rule.value.name
      priority                   = security_rule.value.priority
      direction                  = security_rule.value.direction
      access                     = security_rule.value.access
      protocol                   = security_rule.value.protocol
      source_port_range          =
security_rule.value.source_port_range
      destination_port_range     =
security_rule.value.destination_port_range
      source_address_prefix      =
security_rule.value.source_address_prefix
      destination_address_prefix =
security_rule.value.destination_address_prefix
    }
  }
}
```

# Terraform Registry

Terraform offers a public registry available here :
https://registry.terraform.io/

Using them is relatively simple, and a simple "terraform init" after having configured them in your project allows their use.

The following example does exactly the same thing as in our previous examples using this module :
https://registry.terraform.io/modules/getindata/resource-group/azurerm/latest

```
module "resource-group" {
  source  = "getindata/resource-group/azurerm"
  version = "1.2.1"
  location = "France Central"
  name     = "example-rg-from-module"
}
```

# Vault

```
provider "vault" {
  # Ensure these environment variables are set:
  # VAULT_ADDR: The address of your Vault server
  # VAULT_TOKEN: A token with access to the necessary secrets
}

data "vault_generic_secret" "password" {
  path = "secret/data/azure/db"
}

resource "azurerm_sql_server" "example" {
  name                         = "example-sqlserver"
  resource_group_name          =
azurerm_resource_group.example.name
  location                     =
azurerm_resource_group.example.location
  version                      = "12.0"
  administrator_login          = "sqladmin"
  administrator_login_password =
data.vault_generic_secret.password.data["password"]
}
```

Vault can be used with Terraform to securely manage the secrets or sensitive data needed in your Terraform configurations.

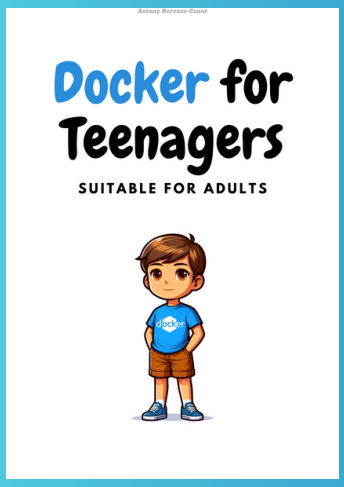
Terraform has a Vault provider that allows reading information from Vault and using it in your Terraform configurations.

# Consul

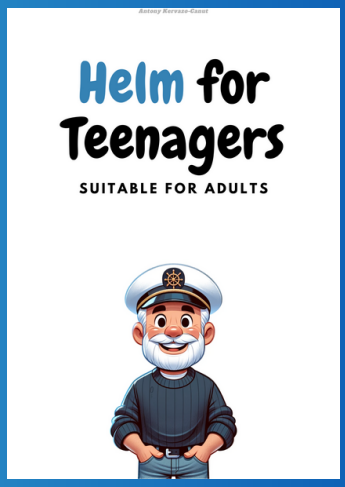
```
provider "consul" {
  address    = "consul.mycompany.com"
  datacenter = "dc1"
}

data "consul_keys" "app_config" {
  key {
    path = "config/myapp/port"
  }
}

resource "azurerm_virtual_machine" "example" {
  name                  = "example-vm"
  location              =
azurerm_resource_group.example.location
  resource_group_name   = azurerm_resource_group.example.name
  network_interface_ids =
[azurerm_network_interface.example.id]
  vm_size               = "Standard_DS1_v2"

  os_profile {
    computer_name  = "hostname"
    admin_username = "adminuser"
  }

  tags = {
    "AppPort" = data.consul_keys.app_config.var.port
  }
}
```

**Consul can be used with Terraform to store configurations or to discover services and IP addresses within your infrastructure.**

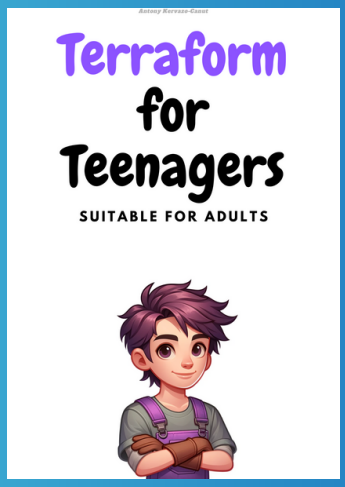
# In the same collection












ANTONYCANUT

ANTONY KERVAZO-CANUT