# Practice M4: Terraform Fundamentals

For this practice we will use an infrastructure like this:

A screenshot of a computer

AI-generated content may be incorrect.

During this practice we will assume that we are working in **Windows** environment. It could be a physical machine or a virtual one. You must have **Vagrant** and **VirtualBox** installed, or at least local instance of **Docker**

All steps can be executed on Linux (the distribution of choice is not that important, but it will be better to stick to some of the well supported distributions) and/or macOS environment as well

## Part 1: Terraform

#### Installation (on Linux)

Open a terminal session and type

**wget https://releases.hashicorp.com/terraform/1.12.2/terraform\_1.12.2\_linux\_amd64.zip -O /tmp/terraform.zip**

**unzip /tmp/terraform.zip -d /tmp**

**sudo mv /tmp/terraform /usr/local/bin**

*For ARM64-based hardware use a different link in the above command:*

*https://releases.hashicorp.com/terraform/1.12.2/terraform\_1.12.2\_linux\_arm64.zip*

Now to test that everything is working as expected type:

**terraform version**

To see what commands are supported type:

**terraform**

Help for a command can be seen by typing:

**terraform -help [command]**

#### Syntax highlighting for Vim (on Linux)

*If not using vim, then skip this block*

First execute:

**mkdir -p ~/.vim/autoload ~/.vim/bundle && \**

**curl -LSso ~/.vim/autoload/pathogen.vim https://tpo.pe/pathogen.vim**

Then edit your **~/.vimrc** file and add the following:

**execute pathogen#infect()**

**syntax on**

**filetype plugin indent on**

As last step install the **vim-terraform** plugin:

**cd ~/.vim/bundle**

**git clone https://github.com/hashivim/vim-terraform.git**

#### Installation (on Windows)

Open a browser tab and navigate to <https://www.terraform.io/downloads.html>

Download the package that corresponds to your version of Windows

Extract package content to a folder of your choice

Include the target folder in the **PATH** environment variable

Open a terminal session

Now to test that everything is working as expected type:

**terraform version**

To see what commands are supported type:

**terraform**

Help for a command can be seen by typing:

**terraform -help [command]**

#### Terraform plugin for VS Code

Knowing that **Visual Studio Code** is a nice multi-platform extensible editor we assume that it will be used during this practice and the ones that follow

We can open **Visual Studio Code**

Switch to the **Extensions** view

Enter the **terraform** term in the search box and hit **Enter**

You will see plenty of extensions

Pick the one you like. One of the most popular ones is the one coming from **HashiCorp** (**HashiCorp Terraform**). There are others popular as well, for example, the one coming from **Microsoft** (**Microsoft Terraform**) which is for Terraform on Azure

Click the **Install** button

After a while, we will have the extension installed

#### Setup the playground

Of course, you can extract the practice archive in a folder of your choice, but it would be better to type all by yourself

Create a folder to accommodate our practice files. For example, **<home folder>\do1\m4\p1**

Open the folder in **VS Code**

#### Let's start

First, we will take a look at how we can interact with a cloud platform and then with a local virtualization solution

Even though only a few are listed here, many more are supported

Do not feel obliged to try each one of the listed below. Instead pick one cloud based and one on-premises

##### Amazon Web Services

Documentation is available here:

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

Create an empty file **main.tf** with the following content:

provider "aws" {

  access\_key = "<ACCESS-KEY>"

  secret\_key = "<SECRET-KEY>"

  region     = "eu-central-1"

}

resource "aws\_instance" "vm1" {

  # Amazon Linux 2023 AMI (HVM) - Kernel 6.1, SSD Volume Type

  ami           = "ami-0229b8f55e5178b65"

  instance\_type = "t2.micro"

}

*NOTE1: Substitute* ***<ACCESS-KEY>*** *and* ***<SECRET-KEY>*** *values with the ones corresponding to a dedicated user in your AWS account. If you do not have, create one. It must have* ***AmazonEC2FullAccess*** *permissions*

*NOTE2: Of course, it is not considered a good practice to store sensitive data like access keys or secret keys in configuration files. We will see later, how we can deal with this. If you are eager to find a way now, then you can use environment variables with an* ***AWS*** *prefix. For example,* ***AWS\_ACCESS\_KEY***

*NOTE3: Or, if you have the AWS CLI installed and configured, you can omit the two lines related to the key*

Now save and exit

To get the list of available images (AMIs) you can follow different approaches. You can use the UI to explore them or use the following commands

For the classic Amazon Linux AMIs, use this one

**aws ec2 describe-images --owners amazon --filters "Name=name,Values=amzn\*" --query 'sort\_by(Images, &CreationDate)[].[ImageId, Name]' --output text**

And for the Amazon Linux 2023 AMIs, use this one

**aws ec2 describe-images --owners amazon --filters "Name=name,Values=al2023\*" --query 'sort\_by(Images, &CreationDate)[].[ImageId, Name]' --output text**

In both cases, you must have your AWS CLI configured upfront

Let's check if we have entered a valid file

Open a terminal (either in **VS Code**, or a separate session) and execute

**terraform validate**

It appears that there is an error – a provider is missing

We can address the error by executing:

**terraform init**

Now if we execute the check again it appears that everything is okay

Let's check how terraform will address our infrastructure:

**terraform plan**

And let's finally create our infrastructure:

**terraform apply**

Once we confirm by entering **yes**, the process of creation will begin

We can go to the **EC2 Dashboard** to examine what we just created

There is also a new file in our working directory – **terraform.tfstate**

Let's examine it. Please note that we should not modify this file manually under any circumstances

It appears that the file is a plain text file in **JSON** format

Compare it with the **terraform.tfstate.backup** file if present

The same information can be obtained with the following command:

**terraform show**

Okay, our instance is not very useful as we don't have a key assigned to it

Let's go to the **EC2 Dashboard** and create a key, for example **terraform-key**

*NOTE: When creating the key, pay attention to the private key format. It should be aligned with the application you plan to use for SSH connections*

Save the key pair file in a folder of your choice and don't forget to adjust its settings (it should be read-only only for you and no one else should have any rights)

Now open the configuration file, go to the resource section, and add (for example, after the **instance\_type** option the following:

key\_name      = "terraform-key"

Where **terraform-key** is the name of the key you created earlier (you should adjust it to match yours)

Save the file

Go to the terminal and type:

**terraform plan**

It appears that because of this change our instance will be destroyed and created again

Apply the changes:

**terraform apply**

Confirm with **yes**

Once our instance is ready, in order to connect to it, we should go again in the **EC2 Dashboard** and check what the public IP address is.

Let's connect to the instance

**ssh -i terraform-key.pem ec2-user@<public-ip-address>**

You should adjust the path to the file (for example **$HOME/.ssh/terraform-key.pem**), username, and the IP address

Execute a command or two and then close the session

We can improve the situation by adding a special instruction, but before doing this, let's explore another option. Type:

**terraform console**

Type **help**

Next enter the following:

**aws\_instance.vm1.id**

So, by typing full resource name plus an attribute we can explore and get useful information

Now, to get the public IP address or DNS name, type:

**aws\_instance.vm1.public\_ip**

**aws\_instance.vm1.public\_dns**

Let's close the console by typing **exit**

It will be nice if we can have this information automatically as a summary instead of typing multiple commands

Let's change again our file by adding near the end the following block:

output "public\_ip" {

  value = aws\_instance.vm1.public\_ip

}

output "public\_dns" {

  value = aws\_instance.vm1.public\_dns

}

Save the file and then execute:

**terraform apply**

Now that we see the public IP address on the terminal, we can try again to connect

There are a few more things we can do. For example, we can improve the formatting of our file, by executing:

**terraform fmt** **-diff=true**

Under **Windows** skip the **-diff=true** part

If we open the file, we will see that indeed it is structured better than before

There is an option to create a visual graph (you will need the **GraphViz** tool) of the resources by executing:

**terraform graph | dot -Tpng -o main.png**

Or ask for specific operation graph, for example, the plan operation

**terraform graph -type=plan | dot -Tpng -o main-plan.png**

We can stop and remove our infrastructure by:

**terraform destroy**

The process will begin after our confirmation

This is an alias to **terraform apply -destroy** , so we can use it instead

##### Azure

Documentation is available here:

<https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs>

Create an empty file **main.tf** with the following content:

provider "azurerm" {

  features {}

  subscription\_id = "<azure\_subscription\_id>"

  tenant\_id       = "<azure\_subscription\_tenant\_id>"

  client\_id       = "<service\_principal\_appid>"

  client\_secret   = "<service\_principal\_password>"

}

resource "azurerm\_resource\_group" "rg" {

  name     = "rg-terraform"

  location = "West Europe"

}

resource "azurerm\_virtual\_network" "vnet" {

  name                = "vnet"

  address\_space       = ["10.0.0.0/16"]

  location            = azurerm\_resource\_group.rg.location

  resource\_group\_name = azurerm\_resource\_group.rg.name

}

resource "azurerm\_subnet" "snet" {

  name                 = "internal"

  resource\_group\_name  = azurerm\_resource\_group.rg.name

  virtual\_network\_name = azurerm\_virtual\_network.vnet.name

  address\_prefixes     = ["10.0.2.0/24"]

}

resource "azurerm\_network\_interface" "vm1nic" {

  name                = "vm1nic"

  location            = azurerm\_resource\_group.rg.location

  resource\_group\_name = azurerm\_resource\_group.rg.name

  ip\_configuration {

    name                          = "internal"

    subnet\_id                     = azurerm\_subnet.snet.id

    private\_ip\_address\_allocation = "Dynamic"

  }

}

resource "azurerm\_linux\_virtual\_machine" "vm1" {

  name                            = "vm1"

  resource\_group\_name             = azurerm\_resource\_group.rg.name

  location                        = azurerm\_resource\_group.rg.location

  size                            = "Standard\_B1s"

  disable\_password\_authentication = "false"

  admin\_username                  = "adminuser"

  admin\_password                  = "TerraformRulez!"

  network\_interface\_ids = [

    azurerm\_network\_interface.vm1nic.id,

  ]

  os\_disk {

    caching              = "ReadWrite"

    storage\_account\_type = "Standard\_LRS"

  }

  source\_image\_reference {

    publisher = "Canonical"

    offer     = "ubuntu-24\_04-lts"

    sku       = "server"

    version   = "latest"

  }

}

*NOTE1: Substitute* ***<SUBSCRIPTION-ID>, <TENANT\_ID>, <CLIENT\_ID>*** *and* ***<CLIENT\_SECRET>*** *values with the ones corresponding to a dedicated user in your Azure account. If you do not have, create one. It must have* ***Contributor*** *role*

*NOTE2: Of course, it is not considered a good practice to store sensitive data like access keys or secret keys in configuration files. We will see later, how we can deal with this. If you are eager to find a way now, then you can use environment variables with an* ***ARM*** *prefix. For example,* ***ARM\_CLIENT\_ID***

To obtain the above, we must execute a few commands

Authenticate the Azure CLI to the Azure cloud

**az login**

If more than one subscription is returned, then we must set which one will be used as default by executing

**az account set --subscription "<subscription-id>"**

We will use the same ID in the file (**main.tf**) as well

Then, we must create a service principal with the following command

**az ad sp create-for-rbac --role="Contributor" --scopes="/subscriptions/<subscription-id>"**

Now, we should have all the required information

Let's fill in all values in the **main.tf** file

Now save and exit

Let's check if we have entered a valid file

Open a terminal (either in **VS Code**, or a separate session) and execute

**terraform validate**

It appears that there is an error – a provider is missing

We can address the error by executing:

**terraform init**

Now if we execute the check again it appears that everything is okay

Let's check how terraform will address our infrastructure:

**terraform plan**

And let's finally create our infrastructure:

**terraform apply**

Once we confirm by entering **yes**, the process of creation will begin

We can go to the **Azure Portal** to examine what we just created

There is also a new file in our working directory – **terraform.tfstate**

Let's examine it. Please note that we should not modify this file manually under any circumstances

It appears that the file is a plain text file in **JSON** format

Same information can be obtained with the following command:

**terraform show**

Okay, our instance is not very useful as we don't have a public IP address assigned to it

Add the following at the end of the file

resource "azurerm\_public\_ip" "vm1pip" {

  name                = "vm1pip"

  resource\_group\_name = azurerm\_resource\_group.rg.name

  location            = azurerm\_resource\_group.rg.location

  sku                 = "Basic"

  allocation\_method   = "Dynamic"

}

And add this line in the network configuration block (in **ip\_configuration** under **azurerm\_network\_interface**):

public\_ip\_address\_id          = azurerm\_public\_ip.vm1pip.id

Save the file

Go to the terminal and type:

**terraform plan**

It appears that because of this change just the network interface of the instance will be changed (of course, beside the creation of a public IP)

Apply the changes:

**terraform apply**

Confirm with **yes**

Once our instance is ready, in order to connect to it, we should go again in the **Azure Portal** and check what the public IP address is

Let's connect to the instance

**ssh adminuser@<public-ip-address>**

Execute a command or two and then close the session

We can improve the situation by adding a special instruction, but before doing this, let's explore another option:

**terraform console**

Type **help**

Next enter the following:

**azurerm\_linux\_virtual\_machine.vm1.id**

**azurerm\_linux\_virtual\_machine.vm1.name**

So, by typing full resource name plus an attribute we can explore and get useful information

Now, to get the public IP address, type:

**azurerm\_linux\_virtual\_machine.vm1.public\_ip\_address**

Hmm, nothing appears *(if it is of type* ***dynamic*** *then it may not get published in the state)*

Let's close the console by typing **exit**

It will be nice if we can have this information automatically as a summary instead of typing multiple commands with debatable outcome

Let's change again our file by adding near the end the following block:

output "public\_ip" {

  value = azurerm\_linux\_virtual\_machine.vm1.public\_ip\_address

}

Save the file and then execute:

**terraform apply**

Now, we may or may not see the public IP address on the terminal, it all depends on the selected Public IP address offer (type). With the Basic Dynamic Public IP addresses, we won’t be able to see the actual address in the output

There are a few more things we can do. For example, we can improve the formatting of our file, by executing:

**terraform fmt -diff=true**

Under **Windows** skip the **-diff=true** part

If we open the file, we will see that indeed it is structured better than before

There is an option to create a visual graph (you will need the **GraphViz** tool) of the resources by executing:

**terraform graph | dot -Tpng -o main.png**

Or ask for specific operation graph, for example, the plan operation

**terraform graph -type=plan | dot -Tpng -o main-plan.png**

We can stop and remove our infrastructure by:

**terraform destroy**

The process will begin after our confirmation

This is an alias to **terraform apply -destroy** , so we can use it instead

##### Google Cloud Platform

Documentation is available here:

<https://registry.terraform.io/providers/hashicorp/google/latest/docs>

Create an empty file **main.tf** with the following content:

terraform {

  required\_providers {

    google = {

      source = "hashicorp/google"

    }

  }

}

provider "google" {

  credentials = file("<FILE\_NAME>")

  project = "<PROJECT\_ID>"

  region  = "europe-north1"

  zone    = "europe-north1-b"

}

resource "google\_compute\_network" "vnet" {

  name = "vnet"

}

resource "google\_compute\_instance" "vm1" {

  name         = "vm1"

  machine\_type = "e2-micro"

  boot\_disk {

    initialize\_params {

      image = "debian-cloud/debian-12"

    }

  }

  network\_interface {

    network = google\_compute\_network.vnet.name

    access\_config {

    }

  }

}

*NOTE1: Substitute* ***<PROJECT\_ID>*** *and* ***<FILE\_NAME>*** *values with the ones corresponding to your situation and Google Cloud Platform setup. If you do not have a service account and credentials file, create one*

Now save and exit

Let's check if we have entered a valid file

Open a terminal (either in **VS Code**, or a separate session) and execute

**terraform validate**

It appears that there is an error – a provider is missing

We can address the error by executing:

**terraform init**

Now if we execute the check again it appears that everything is okay

Let's check how terraform will address our infrastructure:

**terraform plan**

And let's finally create our infrastructure:

**terraform apply**

Once we confirm by entering **yes**, the process of creation will begin

We can go to the **Google Cloud Platform** console to examine what we just created

There is also a new file in our working directory – **terraform.tfstate**

Let's examine it. Please note that we should not modify this file manually under any circumstances

It appears that the file is a plain text file in **JSON** format

Same information can be obtained with the following command:

**terraform show**

Okay, our instance is not very useful as we can see the public IP but don't have any credentials to access it

If we do not have private public key pair, we can create with

**ssh-keygen**

Then, we can add the following block to the compute instance resource

  metadata = {

    ssh-keys = "<USERNAME>:${file("~/.ssh/id\_rsa.pub")}"

  }

Of course, the **<USERNAME>** placeholder should contain our username

Then, we must add one more block. A firewall rule to allow SSH communication

resource "google\_compute\_firewall" "allow\_ssh" {

  name    = "allow-ssh"

  network = google\_compute\_network.vnet.name

  source\_ranges = ["0.0.0.0/0"]

  allow {

    protocol = "tcp"

    ports    = ["22"]

  }

}

Save the file

Go to the terminal and type:

**terraform plan**

It appears that because of this change a few things will be altered

Apply the changes:

**terraform apply**

Confirm with **yes**

Once our instance is ready, we can connect to it

**ssh <public-ip-address>**

Execute a command or two and then close the session

We can improve the situation by adding a special instruction, but before doing this, let's explore another option:

**terraform console**

Type **help**

Next enter the following:

**google\_compute\_instance.vm1.id**

**google\_compute\_instance.vm1.name**

So, by typing full resource name plus an attribute we can explore and get useful information

Now, to get the public IP address, type:

**google\_compute\_instance.vm1.network\_interface[0].access\_config[0].nat\_ip**

Let's close the console by typing **exit**

It will be nice if we can have this information automatically as a summary instead of typing multiple commands with debatable outcome

Let's change again our file by adding near the end the following block:

output "public\_ip" {

  value = google\_compute\_instance.vm1.network\_interface[0].access\_config[0].nat\_ip

}

Save the file and then execute:

**terraform apply**

Now that we see the public IP address on the terminal, we can try again to connect

There are a few more things we can do. For example, we can improve the formatting of our file, by executing:

**terraform fmt -diff=true**

Under **Windows** skip the **-diff=true** part

If we open the file, we will see that indeed it is structured better than before

There is an option to create a visual graph (you will need the **GraphViz** tool) of the resources by executing:

**terraform graph | dot -Tpng -o main.png**

Or ask for specific operation graph, for example, the plan operation

**terraform graph -type=plan | dot -Tpng -o main-plan.png**

We can stop and remove our infrastructure by:

**terraform destroy**

The process will begin after our confirmation

This is an alias to **terraform apply -destroy** , so we can use it instead

##### On-Prem VirtualBox

We can try and spin up a **VirtualBox** virtual machine

Documentation is available here:

<https://registry.terraform.io/providers/shekeriev/virtualbox/latest/docs>

Prepare a folder and navigate to it

Then create a **main.tf** file with the following content:

terraform {

  required\_providers {

    virtualbox = {

      source = "shekeriev/virtualbox"

    }

  }

}

provider "virtualbox" {

  delay      = 60

  mintimeout = 5

}

resource "virtualbox\_vm" "vm1" {

  name   = "debian-12"

  image  = "https://app.vagrantup.com/shekeriev/boxes/debian-12.11/versions/0.1/providers/virtualbox.box"

  cpus   = 1

  memory = "512 mib"

  network\_adapter {

    type           = "hostonly"

    device         = "IntelPro1000MTDesktop"

    host\_interface = "vboxnet1"

    # On Windows use

    # host\_interface = "VirtualBox Host-Only Ethernet Adapter"

  }

}

output "IPAddress" {

  value = element(virtualbox\_vm.vm1.\*.network\_adapter.0.ipv4\_address, 1)

}

Save and close the file

Install the provider with

**terraform init**

Then check the actions that will be taken

**terraform plan**

And finally, deploy the configuration

**terraform apply**

Once done, you can use the IP address to open a connection to the virtual machine

**ssh vagrant@<ip-address>**

Both the user and the password are set to **vagrant**

Execute a command or two and then close the session

Once done exploring, we could create a visual graph (you will need the **GraphViz** tool) of the resources by executing:

**terraform graph | dot -Tpng -o main.png**

Or ask for specific operation graph, for example, the plan operation

**terraform graph -type=plan | dot -Tpng -o main-plan.png**

We can stop and remove our infrastructure by:

**terraform destroy**

## Part 2: Terraform and Docker

Before we begin, we must have a **Docker** instance running

We can use a local one or spin up one with the help of the **Vagrantfile** provided

It spins up a **Docker** host which is listening on all its network interfaces (which makes it accessible from our workstation)

#### Basic infrastructure on Docker

Create a new folder

For example, **<home folder>\do1\m4\p2**

Open the folder in **VS Code** and create an empty file **main.tf**

Add the following:

terraform {

  required\_providers {

    docker = {

      source  = "kreuzwerker/docker"

    }

  }

}

resource "docker\_image" "img-web" {

  name = "shekeriev/terraform-docker:latest"

}

If the **Docker** is not running on our host, we should add the following in the beginning of the file:

provider "docker" {

  host = "tcp://192.168.99.100:2375/"

}

Now we must initialize the environment by executing:

**terraform init**

Then we can execute:

**terraform plan**

If we want to save the plan and later reuse/apply exactly the same plan, we can execute:

**terraform plan -out docker.plan**

And finally:

**terraform apply**

Or if we want to apply a particular plan that we have as a file, we can execute instead:

**terraform apply docker.plan**

After the process ends, we can execute:

**terraform show**

**docker image ls**

If working with a remote **Docker** instance, you may need to add **-H tcp://<docker-ip>**

**docker -H tcp://<docker-ip> image ls**

If using the provided **Vagrantfile** it will become

**docker -H tcp://192.168.99.100 image ls**

Alternatively, you could register a context pointing to the remote **Docker** instance

This could be done by first checking the existing contexts (in order to avoid any conflicts)

**docker context ls**

Then add the remote **Docker** instance

**docker context create docker-99-100 --docker "host=tcp://192.168.99.100:2375"**

Switch to the new context

**docker context use docker-99-100**

And check again

**docker context ls**

If you decide not to use a context, then you should supply the host to all subsequent **docker** commands

Let's edit the **main.tf** file by adding:

resource "docker\_container" "con-web" {

  name = "site"

  image = docker\_image.img-web.image\_id

  ports {

    internal = "80"

    external = "80"

  }

}

Save the file and execute:

**terraform plan**

**terraform apply**

And again, let's check what the result is

**terraform show**

**docker container ls**

Let's open a browser tab and enter <http://localhost:8000> or <http://192.168.99.100>

If **Docker** is running elsewhere, we should adjust the address accordingly

We can clean up by executing:

**terraform destroy**

Then we can check **Docker** as well:

**docker container ls -a**

**docker image ls**

If we destroyed the solution, we could create it again with:

**terraform apply**

Now, let's imagine that we have a solution with many components (resources), and we need to update a few of them, there is a way to do it

We can force a resource to be updated by marking it as taint

Taints are stored in the **Terraform** state

Let’s taint our container:

**terraform taint docker\_container.con-web**

Now if we ask again for the plan, we will see that the container will be recreated

**terraform plan**

We can apply the changes or revert by executing:

**terraform untaint docker\_container.con-web**

Let's ask again for the plan

**terraform plan**

Everything seems to be okay

*Please note, that in the recent versions, this command (****taint****) is considered deprecated and should be avoided. You can find more information about this change here:* [*https://developer.hashicorp.com/terraform/cli/commands/taint*](https://developer.hashicorp.com/terraform/cli/commands/taint)

*The same applies for the* ***untaint*** *command*

Examine the available attributes by

**terraform show**

All those are attributes which values can be displayed after an apply command for example

Okay, let's output some information as a summary. Add to the end of the **main.tf** file:

output "container-id" {

  value = docker\_container.con-web.id

}

output "container-name" {

  value = docker\_container.con-web.name

}

Check and then apply:

**terraform plan**

**terraform apply**

#### Parametrization and modularization

Having a lot of parameters hard coded is not good practice

Instead, we must move them to variables

Let’s extend the **main.tf** file with a few variables:

variable "v\_image" {

  description = "Image"

}

variable "v\_con\_name" {

  description = "Container name"

}

variable "v\_int\_port" {

  description = "Internal port"

}

variable "v\_ext\_port" {

  description = "External port"

}

Now we will substitute all four hardcoded values with a reference to the corresponding variable. For example, **name = "shekeriev/terraform-docker:latest"** will become **name = var.v\_image**

The two resources blocks must look like this

resource "docker\_image" "img-web" {

  name = var.v\_image

}

resource "docker\_container" "con-web" {

  name = var.v\_con\_name

  image = docker\_image.img-web.image\_id

  ports {

    internal = var.v\_int\_port

    external = var.v\_ext\_port

  }

}

Save the file

Ask for the plan with

**terraform plan**

We are asked to enter values for every variable. Hit **Ctrl+C** to break the process

We can override this behavior by adding **default** clauses to each variable

variable "v\_image" {

  description = "Image"

  default = "shekeriev/terraform-docker:latest"

}

variable "v\_con\_name" {

  description = "Container name"

  default = "site"

}

variable "v\_int\_port" {

  description = "Internal port"

  default = 80

}

variable "v\_ext\_port" {

  description = "External port"

  default = 80

}

If we ask again for the plan, no input will be required

Next step would be breaking our big **main.tf** file in parts

Let's create one file called **variables.tf** that will hold the four variables' definitions

And another one called **output.tf** for the two output instructions

Once ready, we can ask again about the plan, or we can destroy and then recreate the whole infrastructure

#### Environment separation

In order to implement environment separation like **dev vs prod** for example, we must add one more variable and modify a little bit all other variables plus the **main.tf** file.

Open **variables.tf** in the main folder and add the following on the top:

variable "mode" {

  description = "mode: prod or dev"

}

Then modify the rest of the variables to become maps, and set values of your choice, like:

variable "v\_ext\_port" {

description = "External port"

type = map

  default = {

    dev = "8080"

    prod = "80"

  }

}

The **variables.tf** file should look like:

variable "mode" {

  description = "mode: prod or dev"

}

variable "v\_image" {

  description = "Image"

  type        = map(any)

  default = {

    dev  = "shekeriev/terraform-docker:dev"

    prod = "shekeriev/terraform-docker:prod"

  }

}

variable "v\_con\_name" {

  description = "Container name"

  type        = map(any)

  default = {

    dev  = "site-dev"

    prod = "site-prod"

  }

}

variable "v\_int\_port" {

  description = "Internal port"

  type        = map(any)

  default = {

    dev  = 80

    prod = 80

  }

}

variable "v\_ext\_port" {

  description = "External port"

  type        = map(any)

  default = {

    dev  = 8080

    prod = 80

  }

}

It is time to adjust the **main.tf**:

terraform {

  required\_providers {

    docker = {

      source = "kreuzwerker/docker"

    }

  }

}

provider "docker" {

  host = "tcp://192.168.99.100:2375/"

}

resource "docker\_image" "img-web" {

  name = lookup(var.v\_image, var.mode)

}

resource "docker\_container" "con-web" {

  name  = lookup(var.v\_con\_name, var.mode)

  image = docker\_image.img-web.image\_id

  ports {

    internal = lookup(var.v\_int\_port, var.mode)

    external = lookup(var.v\_ext\_port, var.mode)

  }

}

Let's test with:

**terraform plan**

**terraform apply**

Next, before we start experimenting with workspaces, let's clean up with:

**terraform destroy**

#### Workspaces

We can have more than one environment up and running. This is handled with workspaces.

First, let's check what workspaces we have currently:

**terraform workspace list**

Now, we can create two – one for **production** and one for **development**:

**terraform workspace new production**

**terraform workspace new development**

If we ask once again for the list of workspaces

**terraform workspace list**

We will see that we have three in total, and that currently selected is the last one we created earlier – **development**

Now, we can create the infrastructure and set dev as mode:

**terraform apply -var 'mode=dev'**

And then switch to the other workspace:

**terraform workspace select production**

And why not spin up a new infrastructure, this time in production mode:

**terraform apply -var 'mode=prod'**

We can use our browser to check both web applications

For dev, go to <http://localhost:8080>

For prod, go to <http://localhost>

Now, we are ready to clean up, but in order to do it according to the books, we must destroy each infrastructure and then the workspaces

So, in order to delete the prod environment, we can execute the following:

**terraform destroy -var 'mode=prod'**

**terraform workspace select development**

**terraform workspace delete production**

So, in order to delete the dev environment, we can execute the following:

**terraform destroy -var 'mode=dev'**

**terraform workspace select default**

**terraform workspace delete development**

#### Variable separation

Create an empty file. For name either set **terraform.tfvars** or an arbitrary name with extension **tfvars**

Move all potentially (not in our case) sensitive information to this file

The content should look like:

v\_image = {

    dev = "shekeriev/terraform-docker:dev"

    prod = "shekeriev/terraform-docker:prod"

}

v\_con\_name = {

    dev = "site-dev"

    prod = "site-prod"

}

v\_int\_port = {

    dev = 80

    prod = 80

}

v\_ext\_port = {

    dev = 8080

    prod = 80

}

From the **variables.tf** file remove all **default** sections

variable "mode" {

  description = "mode: prod or dev"

}

variable "v\_image" {

  description = "Image"

  type = map

}

variable "v\_con\_name" {

  description = "Container name"

  type = map

}

variable "v\_int\_port" {

  description = "Internal port"

  type = map

}

variable "v\_ext\_port" {

  description = "External port"

  type = map

}

Now execute:

**terraform plan**

If your file is with custom name, for example **myvars.tfvars** or is in another folder, then you should extend the command:

**terraform plan -var-file="myvars.tfvars"**

Then we can spin up our infrastructure and check that everything is working as expected

**terraform apply**

And finally, we can clean up everything

**terraform destroy**

## Part 3: Terraform and AWS

For this set of tasks, we can work in a more comfortable environment. For example, we can install **VS Code** with the following plugins - **Terraform** and **Advanced Terraform Snippets Generator**

#### Single file solution

Create a new folder and navigate to it

Create an empty **main.tf** file. Then enter:

provider "aws" {

  access\_key = "<ACCESS-KEY>"

  secret\_key = "<SECRET-KEY>"

  region     = "eu-central-1"

}

resource "aws\_vpc" "do1-vpc" {

  cidr\_block           = "10.10.0.0/16"

  enable\_dns\_hostnames = true

  enable\_dns\_support   = true

  tags = {

    Name = "DO1-VPC"

  }

}

resource "aws\_internet\_gateway" "do1-igw" {

  vpc\_id = aws\_vpc.do1-vpc.id

  tags = {

    Name = "DO1-IGW"

  }

}

resource "aws\_route\_table" "do1-prt" {

  vpc\_id = aws\_vpc.do1-vpc.id

  route {

    cidr\_block = "0.0.0.0/0"

    gateway\_id = aws\_internet\_gateway.do1-igw.id

  }

  tags = {

    Name = "DO1-PUBLIC\_RT"

  }

}

resource "aws\_subnet" "do1-snet" {

  vpc\_id                  = aws\_vpc.do1-vpc.id

  cidr\_block              = "10.10.10.0/24"

  map\_public\_ip\_on\_launch = true

  tags = {

    Name = "DO1-SUB-NET"

  }

}

resource "aws\_route\_table\_association" "do1-prt-assoc" {

  subnet\_id      = aws\_subnet.do1-snet.id

  route\_table\_id = aws\_route\_table.do1-prt.id

}

resource "aws\_security\_group" "do1-pub-sg" {

  name        = "do1-pub-sg"

  description = "DO1 Public SG"

  vpc\_id      = aws\_vpc.do1-vpc.id

  ingress {

    from\_port   = 22

    to\_port     = 22

    protocol    = "tcp"

    cidr\_blocks = ["0.0.0.0/0"]

  }

  ingress {

    from\_port   = 80

    to\_port     = 80

    protocol    = "tcp"

    cidr\_blocks = ["0.0.0.0/0"]

  }

  egress {

    from\_port   = 0

    to\_port     = 0

    protocol    = "-1"

    cidr\_blocks = ["0.0.0.0/0"]

  }

}

resource "aws\_instance" "do1-server" {

  ami                    = "ami-0229b8f55e5178b65"

  instance\_type          = "t2.micro"

  key\_name               = "terraform-key"

  vpc\_security\_group\_ids = [aws\_security\_group.do1-pub-sg.id]

  subnet\_id              = aws\_subnet.do1-snet.id

}

output "public\_ip" {

  value = aws\_instance.do1-server.public\_ip

}

Now we are ready to execute:

**terraform init**

**terraform validate**

**terraform plan**

**terraform apply**

Let's go to **AWS EC2 Dashboard** and check the situation

#### Improve the structure by adding a 2nd machine

Let's add second server

Change the last portion (the last two blocks) of the **main.tf** file to:

resource "aws\_instance" "do1-server" {

  count                  = 2

  ami                    = "ami-0229b8f55e5178b65"

  instance\_type          = "t2.micro"

  key\_name               = "terraform-key"

  vpc\_security\_group\_ids = [aws\_security\_group.do1-pub-sg.id]

  subnet\_id              = aws\_subnet.do1-snet.id

  tags = {

    Name = "do1-server-${count.index + 1}"

  }

}

output "public\_ip" {

  value = aws\_instance.do1-server.\*.public\_ip

}

After we save the file, we can execute:

**terraform validate**

**terraform plan**

**terraform apply**

We can check how the changes to our infrastructure are reflected in **AWS EC2 Dashboard**

#### Implement a sort of high availability

We can improve our solution a little bit by making our infrastructure a kind of highly available

Thus, we will add second subnet in different availability zone and move the second instance there

First, we must get all availability zones:

data "aws\_availability\_zones" "do1-avz" {}

Then we must create a list of sub-nets

variable "do1-cidr" {

    type = list

    default = ["10.10.10.0/24", "10.10.11.0/24"]

}

Then we must change the subnet section to:

resource "aws\_subnet" "do1-snet" {

  count                   = 2

  vpc\_id                  = aws\_vpc.do1-vpc.id

  cidr\_block              = var.do1-cidr[count.index]

  map\_public\_ip\_on\_launch = true

  availability\_zone       = data.aws\_availability\_zones.do1-avz.names[count.index]

  tags = {

    Name = "DO1-SUB-NET-${count.index + 1}"

  }

}

As the next step, we must alter the route table association as well:

resource "aws\_route\_table\_association" "do1-prt-assoc" {

  count          = 2

  subnet\_id      = aws\_subnet.do1-snet.\*.id[count.index]

  route\_table\_id = aws\_route\_table.do1-prt.id

}

And then we must change the **subnet\_id** line in our **do1-server** instances to:

    subnet\_id = element(aws\_subnet.do1-snet.\*.id, count.index)

The above is an alternative way of making the selection we did so far. Instead, we can use the known approach

subnet\_id = aws\_subnet.do1-snet.\*.id[count.index]

Now we are ready to check and apply changes:

**terraform validate**

**terraform plan**

**terraform apply**

#### Provision the machines

Now it is time to make those machines do something meaningful

Let's make them web servers.

For this purpose, we will create a **provision.sh** file with the following content:

#!/bin/bash

sudo dnf install -y nginx

sudo systemctl start nginx

sudo systemctl enable nginx

Then we will edit the file **main.tf**

We must extend the instance provision section (after the tags) with:

  provisioner "file" {

    source      = "./provision.sh"

    destination = "/tmp/provision.sh"

    connection {

      type        = "ssh"

      user        = "ec2-user"

      private\_key = file("terraform-key.pem")

      host        = self.public\_ip

    }

  }

  provisioner "remote-exec" {

    inline = [

      "chmod +x /tmp/provision.sh",

      "/tmp/provision.sh"

    ]

    connection {

      type        = "ssh"

      user        = "ec2-user"

      private\_key = file("terraform-key.pem")

      host        = self.public\_ip

    }

  }

It is time to execute again:

**terraform validate**

Next, to force the provisioner execution, we can adopt this approach

**terraform apply -replace="aws\_instance.do1-server[0]"**

**terraform apply -replace="aws\_instance.do1-server[1]"**

Now if we visit each of the public IP addresses, we should see the **NGINX** welcome message

#### Create output.tf and variables.tf files

Let's start backwards. First, we will create **output.tf** file and **move** there the corresponding lines from the **main.tf** file:

output "public\_ip" {

  value = aws\_instance.do1-server.\*.public\_ip

}

Now we must create the **variables.tf** file and put there:

# Variables

# Some sensitive information

variable "v-access-key" {}

variable "v-secret-key" {}

# Shareable information

variable "v-ami-image" {

  description = "AMI image"

  default     = "ami-0229b8f55e5178b65"

}

variable "v-instance-type" {

  description = "EC2 instance type"

  default     = "t2.micro"

}

variable "v-instance-key" {

  description = "Instance key"

  default     = "terraform-key"

}

variable "v-count" {

  description = "Resource count"

  default     = "2"

}

data "aws\_availability\_zones" "do1-avz" {}

variable "do1-cidr" {

  type    = list(any)

  default = ["10.10.10.0/24", "10.10.11.0/24"]

}

Don't forget to **remove** the following from the **main.tf** file

data "aws\_availability\_zones" "do1-avz" {}

variable "do1-cidr" {

  type    = list(any)

  default = ["10.10.10.0/24", "10.10.11.0/24"]

}

Now we will create a **terraform.tfvars** file to hold our sensitive data:

# Secret information :)

v-access-key = "<ACCESS-KEY>"

v-secret-key = "<SECRET-KEY>"

Then we should substitute the following in the **main.tf** file:

  access\_key = "<ACCESS-KEY>"

becomes

  access\_key = var.v-access-key

And this

secret\_key = "<SECRET-KEY>"

becomes

  secret\_key = var.v-secret-key

Then all three occurences of

    count = 2

must become

    count = var.v-count

And finally, some corrections of the instances. The lines

  ami                    = "ami-0229b8f55e5178b65"

  instance\_type          = "t2.micro"

  key\_name               = "terraform-key"

must become

ami                    = var.v-ami-image

  instance\_type          = var.v-instance-type

  key\_name               = var.v-instance-key

Finally, save the file

Check the validity with

**terraform validate**

If everything went okay, then execute the command:

**terraform plan**

Should return that there is nothing to change

If we want to see the changes in the output routine, then we can execute either:

**terraform apply**

or

**terraform refresh**

And finally, we can clean up by executing:

**terraform destroy**

That is all for this practice 😊