Practice M1: Infrastructure as Code

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 in Linux (the distribution of choice is not that important but 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/1.4.4/terraform_1.4.4_linux_amd64.zip -O /tmp/terraform.zip
```

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

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

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

Let us pick the one coming from **HashiCorp** - **HashiCorp Terraform**

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>\do2\m1\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 2 AMI (HVM) - Kernel 5.10, SSD Volume Type
  ami           = "ami-0dcc0ebde7b2e00db"
  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***

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 **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

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 is the public IP address

Let's connect to the instance

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

You should adjust the path to the file, 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 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`

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     = "UbuntuServer"
    sku       = "18.04-LTS"
    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 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
  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 is the public IP address

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 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 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`

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-11"
    }
  }

  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 in 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 the 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 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`

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/terra-farm/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-11"
  image  = "https://app.vagrantup.com/shekeriev/boxes/debian-
11/versions/0.2/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

Once done exploring, destroy the machine with

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>\do2\m1\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 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

The above rule applies for 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 is the result

terraform show

docker -H tcp://192.168.99.100 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 -H tcp://192.168.99.100 container ls -a

docker -H tcp://192.168.99.100 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*

*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 a good practice

Instead, we must move them to variables

Lets 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** clause 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 for the plan, or we can destroy and then recreate the whole infrastructure

Further improvements (modules)

It would be better to clean up a little bit, so we will execute

`terraform destroy`

Or if we are tired of entering **yes** every time, we can execute:

`terraform destroy -auto-approve`

We can place each resource in a separate module

To prepare the file structure, create two sub-directories named **image** and **container**

Create a set of empty files **main.tf**, **variables.tf**, and **output.tf** in both folders

Next, we can copy the corresponding information from the files in the main (project root) folder to the modules

First, we will take care of the image module, but with a few changes – we will remove the default value, and we will add an output clause for the image name. So, the **output.tf** file should look like:

```
output "image_out" {
  value = docker_image.img-web.name
}
```

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

```
variable "v_image" {
  description = "Image"
}
```

The final version of the **main.tf** file for the **image** module should be

```
terraform {
  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
    }
  }
}

provider "docker" {
  host = "tcp://192.168.99.100:2375/"
}

resource "docker_image" "img-web" {
  name = var.v_image
}
```

Before we move on, let's test the image module alone

Make sure that you execute the following commands in the **image** folder

`terraform init`

`terraform plan`

`terraform apply`

`terraform destroy`

We are ready to create our **container** module

This time we will migrate the container resource, and all variables (including the one for the image)

Again, we will remove the default values. So, the **variable.tf** will look like

```
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"
}
```

The final version of the **main.tf** file for the **container** module should be

```
terraform {
  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
    }
  }
}

provider "docker" {
  host = "tcp://192.168.99.100:2375/"
}

resource "docker_container" "con-web" {
  name = var.v_con_name
  image = var.v_image
  ports {
    internal = var.v_int_port
    external = var.v_ext_port
  }
}
```

And the **output.tf** will have the following content

```
output "container-id" {
  value = docker_container.con-web.id
}

output "container-name" {
  value = docker_container.con-web.name
}
```

Once ready we can ask for the plan just to be sure that everything is working

`terraform init`

terraform plan

terraform apply

terraform destroy

Don't forget to initialize before asking for the plan.

As a final step we must alter our **main.tf** file (the one in the base folder). It should contain only:

```
terraform {
  required_providers {
    docker = {
      source  = "kreuzwerker/docker"
    }
  }
}

provider "docker" {
  host = "tcp://192.168.99.100:2375/"
}

module "image" {
  source = "./image"
  v_image = var.v_image
}

module "container" {
  source = "./container"
  v_image = module.image.image_out
  v_con_name = var.v_con_name
  v_int_port = var.v_int_port
  v_ext_port = var.v_ext_port
}
```

And ensure that our **output.tf** file is empty but the **variables.tf** is intact

Now we can execute:

terraform init

terraform plan

terraform apply

docker -H tcp://192.168.99.100 container ls

And finally, we can open a tab in our browser and point it to http://localhost

Let's clean everything once again

terraform destroy

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
    default = {
      dev = "shekeriev/terraform-docker:dev"
      prod = "shekeriev/terraform-docker:prod"
    }
}
variable "v_con_name" {
  description = "Container name"
  type = map
    default = {
      dev = "site-dev"
      prod = "site-prod"
    }
}
variable "v_int_port" {
  description = "Internal port"
  type = map
    default = {
      dev = 80
      prod = 80
    }
}
variable "v_ext_port" {
  description = "External port"
  type = map
    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/"
}

module "image" {
  source = "./image"
  v_image = lookup(var.v_image, var.mode)
}

module "container" {
  source = "./container"
  v_image = module.image.image_out
  v_con_name = lookup(var.v_con_name, var.mode)
  v_int_port = lookup(var.v_int_port, var.mode)
  v_ext_port = 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 new a 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" "do2-vpc" {
  cidr_block = "10.10.0.0/16"
  enable_dns_hostnames = true
  enable_dns_support = true
  tags = {
      Name = "DO2-VPC"
  }
}
```

```
resource "aws_internet_gateway" "do2-igw" {
    vpc_id = aws_vpc.do2-vpc.id
    tags = {
        Name = "DO2-IGW"
    }
}

resource "aws_route_table" "do2-prt" {
    vpc_id = aws_vpc.do2-vpc.id
    route {
        cidr_block = "0.0.0.0/0"
        gateway_id = aws_internet_gateway.do2-igw.id
    }
    tags = {
        Name = "DO2-PUBLIC_RT"
    }
}

resource "aws_subnet" "do2-snet" {
    vpc_id = aws_vpc.do2-vpc.id
    cidr_block = "10.10.10.0/24"
    map_public_ip_on_launch = true
    tags = {
        Name = "DO2-SUB-NET"
    }
}

resource "aws_route_table_association" "do2-prt-assoc" {
    subnet_id = aws_subnet.do2-snet.id
    route_table_id = aws_route_table.do2-prt.id
}

resource "aws_security_group" "do2-pub-sg" {
    name = "do2-pub-sg"
    description = "DO2 Public SG"
    vpc_id = aws_vpc.do2-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" "do2-server" {
    ami = "ami-0dcc0ebde7b2e00db"
    instance_type = "t2.micro"
    key_name = "terraform-key"
    vpc_security_group_ids = [aws_security_group.do2-pub-sg.id]
    subnet_id = aws_subnet.do2-snet.id
}

output "public_ip" {
  value = aws_instance.do2-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" "do2-server" {
    count = 2
    ami = "ami-0dcc0ebde7b2e00db"
    instance_type = "t2.micro"
    key_name = "terraform-key"
    vpc_security_group_ids = [aws_security_group.do2-pub-sg.id]
    subnet_id = aws_subnet.do2-snet.id
    tags = {
        Name = "do2-server-${count.index + 1}"
    }
}
output "public_ip" {
  value = aws_instance.do2-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 a little bit our solution 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" "do2-avz" {}
```

Then we must create a list of sub-nets

```
variable "do2-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" "do2-snet" {
    count = 2
    vpc_id = aws_vpc.do2-vpc.id
    cidr_block = var.do2-cidr[count.index]
    map_public_ip_on_launch = true
    availability_zone = data.aws_availability_zones.do2-avz.names[count.index]
    tags = {
        Name = "DO2-SUB-NET-${count.index + 1}"
    }
}
```

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

```
resource "aws_route_table_association" "do2-prt-assoc" {
    count = 2
    subnet_id = aws_subnet.do2-snet.*.id[count.index]
    route_table_id = aws_route_table.do2-prt.id
}
```

And then we must change the **subnet_id** line in our **do2-server** instances to:

```
    subnet_id = element(aws_subnet.do2-snet.*.id, count.index)
```

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

```
    subnet_id = aws_subnet.do2-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 amazon-linux-extras install -y nginx1
sudo systemctl start nginx
sudo systemctl enable nginx
```

Then we fill edit 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 this approach

terraform apply -replace="aws_instance.do2-server[0]"

terraform apply -replace="aws_instance.do2-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.do2-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-0dcc0ebde7b2e00db"
}
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" "do2-avz" {}
variable "do2-cidr" {
    type = list
    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" "do2-avz" {}
variable "do2-cidr" {
    type = list
    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
```

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 instance. The lines

```
    ami = "ami-0dcc0ebde7b2e00db"
    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`

Use external modules

There is plenty of existing **AWS** modules, that can simplify our code

Here we will use just **two** out of **more than several thousand** modules

All can be examined here: https://registry.terraform.io

Create a new folder

Create a new **main.tf** file

Enter the following to initialize the provider and gather some data:

```
# Configure the provider. This can be omitted if matches the configuration made with
"aws configure"
provider "aws" {
  access_key = "<ACCESS-KEY>"
  secret_key = "<SECRET-KEY>"
  region     = "eu-central-1"
}

# Collect and store data about the default VPC
data "aws_vpc" "default" {
  default = true
}

# Collect and store data about the subnets in the default VPC
data "aws_subnets" "all" {
```

```
  filter {
    name   = "vpc-id"
    values = [data.aws_vpc.default.id]
  }
}

# Get the latest AMI with Amazon Linux 2
data "aws_ami" "amazon_linux" {
  most_recent = true
  owners = ["amazon"]
  filter {
    name = "name"
    values = [
      "amzn2-ami-hvm-*-x86_64-gp2",
    ]
  }
  filter {
    name = "owner-alias"
    values = [
      "amazon",
    ]
 }
}
```

Next, we can add the block for the first module – for creating the security group:

```
# Invoke the Security Group module and create one with a few rules
module "security_group" {
  source      = "terraform-aws-modules/security-group/aws"
  name        = "do2-aws-modules-sg"
  description = "Security group made using an AWS module"
  vpc_id      = data.aws_vpc.default.id
  ingress_cidr_blocks = ["0.0.0.0/0"]
  ingress_rules       = ["http-80-tcp", "all-icmp", "ssh-tcp"]
  egress_rules        = ["all-all"]
}
```

And finally, we can add the block for the second module – for creating the EC2 instance:

```
# Invoke the EC2 module and create an instance
module "ec2" {
  source = "terraform-aws-modules/ec2-instance/aws"
  name                        = "do2-aws-modules-ec2"
  ami                         = data.aws_ami.amazon_linux.id
  instance_type               = "t2.micro"
  key_name                    = "terraform-key"
  subnet_id                   = tolist(data.aws_subnets.all.ids)[0]
  vpc_security_group_ids      = [module.security_group.security_group_id]
  associate_public_ip_address = true
  user_data                   = file("./nginx.sh")
}
```

Now we should save the file

*Note that the user you are using should have the **AmazonSSMReadOnlyAccess** policy attached or at least it should be authorized to perform **ssm:GetParameter** on the **arn:aws:ssm:eu-central-1::parameter/aws/service/ami-amazon-linux-latest/amzn2-ami-hvm-x86_64-gp2** resource*

Let's create another file named **nginx.sh** with the following content:

```
#!/bin/sh
sudo amazon-linux-extras install -y nginx1
sudo systemctl start nginx
sudo systemctl enable nginx
echo '<h1>Hello from NGINX running on AWS EC2 instance</h1>' >
/usr/share/nginx/html/index.html
```

And the final part is to create a file **output.tf** with the following content:

```
output public_ip {
  description = "Public IP"
  value = module.ec2.public_ip
}
output public_dns {
  description = "Public DNS"
  value = module.ec2.public_dns
}
```

Now we are ready to execute:

terraform init

terraform validate

terraform plan

terraform apply

And then using the public IP or DNS, we can check the result in our browser

Finally, we can clean up everything with:

terraform destroy