# Practice M1: Infrastructure as Code

During this practice we will assume that we are working in Linux environment. It could be a physical machine or a virtual one. The distribution of choice is not that important but will be better to stick to some of the well supported distributions. You must have Vagrant and VirtualBox installed, or at least local instance of Docker.

All steps can be executed in Windows and/or macOS environment as well.

## Part 1: Terraform

#### Installation

* Open a terminal session and type

wget https://releases.hashicorp.com/terraform/0.11.10/terraform\_0.11.10\_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]

#### Syntaxis highlighting for Vim

* 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

#### 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.

* Execute the following to prepare the folder structure:

mkdir -p ~/DOF/M1/M1-{1,2,3}

* Now go to folder **~/DOF/M1/M1-1**

#### Let’s start

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

provider "aws" {

access\_key = "AKIAIGOPFHF6K5UPBZVQ"

secret\_key = "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"

region = "eu-central-1"

}

resource "aws\_instance" "M1-1" {

ami = "ami-dd3c0f36"

instance\_type = "t2.micro"

}

* *NOTE: Substitute* ***access\_key*** *and* ***secret\_key*** *values with the ones corresponding to a dedicated user in your AWS account*
* Now save and exit
* Let’s check if we have entered a valid file:

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.state**. Let’s examine it
* 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
* Now open the file and add the following:

key\_name = "terraform-key"

* Save and exit the file. Then type:

terraform plan

* It appears that because of this change our instance will be destroyed and created again
* Apply the changes:

terraform apply

* In order to connect we should go again in the EC2 Dashboard and check what is the public IP
* Let’s connect and then close the connection
* 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.M1-1.id

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

aws\_instance.M1-1.public\_ip

aws\_instance.M1-1.public\_dns

* Let’s close the console by typing exit
* It will be nice if we can have this information as a summary. Let’s change again our file by adding:

output "Public IP" {

value = "${aws\_instance.M1-1.public\_ip}"

}

output "Public DNS" {

value = "${aws\_instance.M1-1.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 few more things we can do. For example, we can improve the formatting of our file, by executing:

terraform fmt -diff=true

* 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 of the resources by executing:

terraform graph | dot -Tpng > main.png

* We can stop and remove our infrastructure by:

terraform destroy

* The process will begin after our confirmation. We can skip it by adding --force at the end

## 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.

#### Basic infrastructure on Docker

* Let’s go to folder **~/DOF/M1/M1-2** and create an empty file **main.tf**
* Add the following:

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://192.168.199.100:2376/"

}

* 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 end, we can execute:

terraform show

docker image ls

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

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

name = "site"

image = "${docker\_image.img-web.latest}"

ports {

internal = "80"

external = "80"

}

}

* Save and exit. Then execute:

terraform plan

terraform apply

* And again, let’s check what is the result

terraform show

docker container ps

* Let’s open a browser tab and enter <http://localhost> . 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 ps -a

docker image ls

* If we destroyed the solution, we can create it again with:

terraform apply

* Now, let’s imagine that we have a solution with many components (resources), and we need to update few of them, there is a way to so it. We can force a resource to be updated by marking it as taint:

terraform taint docker\_container.con-web

* Now if we ask again for the plan, we will see that the container will be recreated
* We can apply the changes or revert by executing:

terraform untaint docker\_container.con-web

* Let’s ask again for the plan. Everything seems to be okay
* 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 **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

* Because having a lot of parameters hard-coded is not a good practice, we can move them to variables. Let’s extend the **main.tf** file with 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}"**
* If we save the file and ask for the plan, we will be asked to enter values for every variable. We can override this behavior by adding **default** clause to each variable. If ask gain 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 variables definitions, and another one called **outputs.tf** for the output instructions
* Once ready, we can ask again for the plan, or we can destroy and then recreate the whole infrastructure

#### Further improvements

* 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 we must execute:

mkdir {image,container}

touch {image,container}/{main.tf,variables.tf,output.tf}

* 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 few changes – we will remove the default value, and we will add an output clause for the image name:

output "image\_out" {

value = "${docker\_image.img-web.name}"

}

* Before we move on, let’s test the image module alone:

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. Once ready we can ask for the plan just to be sure that everything is working. Don’t forget to initialize before asking for the plan.
* As final step we must alter our **main.tf** file. It should contain only:

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 **outputs.tf** file is empty
* Now we can execute:

terraform init

terraform plan

terraform apply

docker container ps

* And finally, we can open a tab in our browser and point it to <http://localhost>
* Let’s clean everything once again

#### 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** and add the following:

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 = "8000"

prod = "80"

}

}

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

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, 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
* 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, we can execute the following:

terraform destroy -var 'mode=prod'

terraform workspace select development

terraform destroy -var 'mode=dev'

terraform workspace select default

terraform workspace delete development

terraform workspace delete production

#### 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. The content should look like:

v\_image {

prod = "shekeriev/terraform-docker:prod"

dev = "shekeriev/terraform-docker:dev"

}

v\_con\_name {

prod = "site-prod"

dev = "site-dev"

}

v\_int\_port {

prod = "80"

dev = "80"

}

v\_ext\_port {

prod = "80"

dev = "8000"

}

* From the **variables.tf** file remove all **default** sections
* 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
* And finally, we can clean up everything

## 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

* Go to folder **M1-3** and create an empty **main.tf** file. Then enter:

provider "aws" {

access\_key = "AKIAIGOPFHF6K5UPBZVQ"

secret\_key = "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"

region = "eu-central-1"

}

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

cidr\_block = "10.10.0.0/16"

enable\_dns\_hostnames = true

enable\_dns\_support = true

tags {

Name = "DOF-VPC"

}

}

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

vpc\_id = "${aws\_vpc.dof-vpc.id}"

tags {

Name = "DOF-IGW"

}

}

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

vpc\_id = "${aws\_vpc.dof-vpc.id}"

route {

cidr\_block = "0.0.0.0/0"

gateway\_id = "${aws\_internet\_gateway.dof-igw.id}"

}

tags {

Name = "DOF-PUBLIC\_RT"

}

}

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

vpc\_id = "${aws\_vpc.dof-vpc.id}"

cidr\_block = "10.10.10.0/24"

map\_public\_ip\_on\_launch = true

tags {

Name = "DOF-SUB-NET"

}

}

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

subnet\_id = "${aws\_subnet.dof-snet.id}"

route\_table\_id = "${aws\_route\_table.dof-prt.id}"

}

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

name = "dof-pub-sg"

description = "SOF Public SG"

vpc\_id = "${aws\_vpc.dof-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" "dof-server" {

ami = "ami-dd3c0f36"

instance\_type = "t2.micro"

key\_name = "terraform-key"

vpc\_security\_group\_ids = ["${aws\_security\_group.dof-pub-sg.id}"]

subnet\_id = "${aws\_subnet.dof-snet.id}"

}

output "public\_ip" {

value = "${aws\_instance.dof-server.public\_ip}"

}

* Now we are ready to execute:

terraform init

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 of the **main.tf** file to:

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

count = 2

ami = "ami-dd3c0f36"

instance\_type = "t2.micro"

key\_name = "terraform-key"

vpc\_security\_group\_ids = ["${aws\_security\_group.dof-pub-sg.id}"]

subnet\_id = "${aws\_subnet.dof-snet.id}"

tags {

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

}

}

output "public\_ip" {

value = "${aws\_instance.dof-server.\*.public\_ip}"

}

* After we save the file, we can execute:

terraform plan

terraform apply

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

#### Implement sort of high-availability

* We can improve a little bit our solution by making it 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" "dof-avz" {}

* Then we must create a list of sub-nets

variable "dof-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" "dof-snet" {

count = 2

vpc\_id = "${aws\_vpc.dof-vpc.id}"

cidr\_block = "${var.dof-cidr[count.index]}"

map\_public\_ip\_on\_launch = true

availability\_zone = "${data.aws\_availability\_zones.dof-avz.names[count.index]}"

tags {

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

}

}

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

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

# subnet\_id = "${aws\_subnet.dof-snet.id}"

count = 2

subnet\_id = "${aws\_subnet.dof-snet.\*.id[count.index]}"

route\_table\_id = "${aws\_route\_table.dof-prt.id}"

}

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

subnet\_id = "${element(aws\_subnet.dof-snet.\*.id, count.index)}"

* Now we are ready to check and apply changes:

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 yum install -y epel-release

sudo yum install -y nginx

sudo systemctl start nginx

sudo systemctl enable nginx

* Then we fill edit file **main.tf**. We must extend the instance provision section with:

provisioner "file" {

source = "./provision.sh"

destination = "/tmp/provision.sh"

connection {

type = "ssh"

user = "centos"

private\_key = "${file("/home/dimitar/Downloads/terraform-key.pem")}"

}

}

provisioner "remote-exec" {

inline = [

"chmod +x /tmp/provision.sh",

"/tmp/provision.sh"

]

connection {

type = "ssh"

user = "centos"

private\_key = "${file("/home/dimitar/Downloads/terraform-key.pem")}"

}

}

* It is time to execute again:

terraform plan

terraform apply

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

#### Create outputs.tf and variables.tf files

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

# Output useful information

output "public\_ip" {

value = "${aws\_instance.dof-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-dd3c0f36"

}

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" "dof-avz" {}

variable "dof-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 = "AKIAIGOPFHF6K5UPBZVQ"

v-secret-key = "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"

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

access\_key = "AKIAIGOPFHF6K5UPBZVQ"

becomes

access\_key = "${var.v-access-key}"

This

secret\_key = "panJ9gt+8L1rr9ZrT+CehpMNDu2aDQLbrZsePYph"

becomes

secret\_key = "${var.v-secret-key}"

* Then all

count = 2

become

count = "${var.v-count}"

* And finally, some corrections of the instance

ami = "ami-dd3c0f36"

instance\_type = "t2.micro"

key\_name = "terraform-key"

become

ami = "${var.v-ami-image}"

instance\_type = "${var.v-instance-type}"

key\_name = "${var.v-instance-key}"

* If everything wen okay 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 600. All can be examined here: <https://registry.terraform.io>

* Go to folder **M1-4** and create a **main.tf** file and 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" {

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\_subnet\_ids" "all" {

vpc\_id = "${data.aws\_vpc.default.id}"

}

# Get the latest AMI with Amazon Linux

data "aws\_ami" "amazon\_linux" {

most\_recent = true

filter {

name = "name"

values = [

"amzn-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 few rules

module "security\_group" {

source = "terraform-aws-modules/security-group/aws"

version = "2.7.0"

name = "dof-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"

instance\_count = 1

name = "dof-aws-modules-ec2"

ami = "${data.aws\_ami.amazon\_linux.id}"

instance\_type = "t2.micro"

key\_name = "dof-aws-modules"

subnet\_id = "${element(data.aws\_subnet\_ids.all.ids, 0)}"

vpc\_security\_group\_ids = ["${module.security\_group.this\_security\_group\_id}"]

associate\_public\_ip\_address = true

user\_data = "${file("./nginx.sh")}"

}

* Now we should save and exit
* Let’s create a file **nginx.sh** with the following content:

#!/bin/sh

yum install -y nginx

chkconfig nginx on

service nginx start

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 **outputs.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 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