**Terraform for AWS Environment**

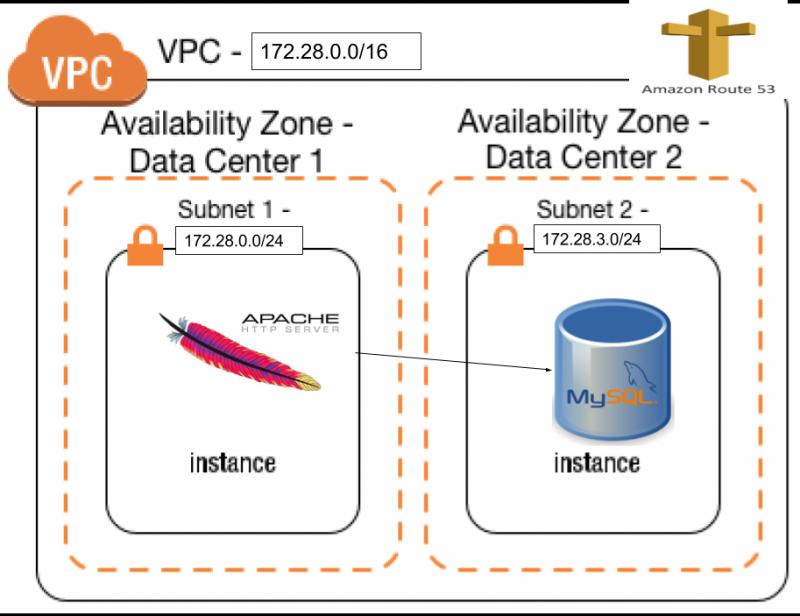
1. Introduction
   1. Prerequisites
2. The files structure
   1. variables.tf
   2. network.tf
   3. routing-and-network.tf
   4. subnets.tf
   5. dns-and-dhcp.tf
   6. ec2-machines.tf
3. The database machine
4. The webapp machine
5. Running the terraform and connect to the application

**Related Guides**

1. [A complete AWS environment with Terraform](https://linuxacademy.com/cp/socialize/index/type/community_post/id/13922)
2. [Automating Terraform with Jenkins and AWS CodeCommit](https://linuxacademy.com/cp/socialize/index/type/community_post/id/18753)

**Introduction**

The purpose of this article is to show a full AWS environment built using the Terraform automation. We will create everything you need from scratch: VPC, subnets, routes, security groups, an EC2 machine with MySQL installed inside a private network, and a webapp machine with Apache and its PHP module in a public subnet. The webapp machine reads a table in the database and shows the result.



**Prerequisites**

There are only 2 prerequisites:

1. Terraform installed.You can find the download page of the latest version here: https://www.terraform.io/downloads.html. At the moment the 0.78 version is available. In order to install it in Linux these 4 commands are enough:

|  |
| --- |
| wget <https://releases.hashicorp.com/terraform/0.7.8/terraform_0.7.8_linux_amd64.zip>  unzip terraform\_0.7.8\_linux\_amd64.zip  sudo mv terraform /usr/bin/ |

1. If you want to log in to the machines, you need to have an AWS pem key already created in the region of your choice and downloaded on your machine.

**The files structure**

Terraform elaborates all the files inside the working directory so it does not matter if everything is contained in a single file or divided into many, although it is convenient to organize the resources in logical groups and split them into different files. Let’s take a look at how we can do this effectively:

**variables.tf**

|  |
| --- |
| variable "region" {  default = "us-west-2"  }  variable "AmiLinux" {  type = "map"  default = {  us-east-1 = "ami-b73b63a0"  us-west-2 = "ami-5ec1673e"  eu-west-1 = "ami-9398d3e0"  }  description = "I add only 3 regions (Virginia, Oregon, Ireland) to show the map feature but you can add all the r"  }  variable "aws\_access\_key" {  default = "”  description = "the user aws access key"  }  variable "aws\_secret\_key" {  default = "”  description = "the user aws secret key"  }  variable "vpc-fullcidr" {  default = "172.28.0.0/16"  description = "the vpc cdir"  }  variable "Subnet-Public-AzA-CIDR" {  default = "172.28.0.0/24"  description = "the cidr of the subnet"  }  variable "Subnet-Private-AzA-CIDR" {  default = "172.28.3.0/24"  description = "the cidr of the subnet"  }  variable "key\_name" {  default = ""  description = "the ssh key to use in the EC2 machines"  }  variable "DnsZoneName" {  default = "saifulcce.internal"  description = "the internal dns name"  } |

All variables are defined in the variables.tf file. Before you run the the “terraform apply” command, you need to insert your access and secret keys. If you also want to log into the EC2 machine, make sure you fill in the key name as well.

Every variable is of type String, except for the AmiLinux. This particular variable is a map and depends on the content of the region variable. You can add the region you wish to use in the map using the ami-id of the AWS Linux distribution.

**network.tf**

|  |
| --- |
| provider "aws" {  access\_key = "${var.aws\_access\_key}"  secret\_key = "${var.aws\_secret\_key}"  region = "${var.region}"  }  resource "aws\_vpc" "terraformmain" {  cidr\_block = "${var.vpc-fullcidr}"  #### this 2 true values are for use the internal vpc dns resolution  enable\_dns\_support = true  enable\_dns\_hostnames = true  tags {  Name = "My terraform vpc"  }  } |

In the network.tf file, we set up the provider for AWS and the VPC declaration. Together with the Route53 configuration, the option specified for the vpc creation enables an internal name resolution for our VPC. As you may be aware, Terraform can be used to build infrastructures for many environments, such as AWS, Azure, Google Cloud, VMware, and many others. A full list is available here:  <https://www.terraform.io/docs/providers/index.html> . In this article, we are using AWS as the provider.

**routing-and-network.tf**

|  |
| --- |
| # Declare the data source  data "aws\_availability\_zones" "available" {}  /\* EXTERNAL NETWORG , IG, ROUTE TABLE \*/  resource "aws\_internet\_gateway" "gw" {  vpc\_id = "${aws\_vpc.terraformmain.id}"  tags {  Name = "internet gw terraform generated"  }  }  resource "aws\_network\_acl" "all" {  vpc\_id = "${aws\_vpc.terraformmain.id}"  egress {  protocol = "-1"  rule\_no = 2  action = "allow"  cidr\_block = "0.0.0.0/0"  from\_port = 0  to\_port = 0  }  ingress {  protocol = "-1"  rule\_no = 1  action = "allow"  cidr\_block = "0.0.0.0/0"  from\_port = 0  to\_port = 0  }  tags {  Name = "open acl"  }  }  resource "aws\_route\_table" "public" {  vpc\_id = "${aws\_vpc.terraformmain.id}"  tags {  Name = "Public"  }  route {  cidr\_block = "0.0.0.0/0"  gateway\_id = "${aws\_internet\_gateway.gw.id}"  }  }  resource "aws\_route\_table" "private" {  vpc\_id = "${aws\_vpc.terraformmain.id}"  tags {  Name = "Private"  }  route {  cidr\_block = "0.0.0.0/0"  nat\_gateway\_id = "${aws\_nat\_gateway.PublicAZA.id}"  }  }  resource "aws\_eip" "forNat" {  vpc = true  }  resource "aws\_nat\_gateway" "PublicAZA" {  allocation\_id = "${aws\_eip.forNat.id}"  subnet\_id = "${aws\_subnet.PublicAZA.id}"  depends\_on = ["aws\_internet\_gateway.gw"]  } |

When you start from scratch, you need to attach an internet gateway to your VPC and define a network ACL. There aren’t restriction at network ACL level because the restriction rules will be enforced by security group.

As you can see, there are two routing tables: one for public access, and the other one for private access. In our case, we also need to have access to the internet from the database machine since we use it to install MySQL Server. We will use the AWS NAT Gateway in order to increase our security and be sure that there aren’t incoming connections coming from outside the database. As you can see, defining a NAT gateway is pretty easy since it consists of only four lines of code. It is important, though, to deploy it in a public subnet and associate an elastic ip to it. The depends\_on allows us to avoid errors and create the NAT gateway only after the internet gateway is in the available state.

One thing worth noting is that the data called aws\_availability\_zones provide the correct name of the availability zones in the chosen region. This way we don’t need to add letters to the region variable and we can avoid mistakes. For example, the North Virginia region where region b does not exist, and in other regions where there are 2 or 4 AZs .

**subnets.tf**

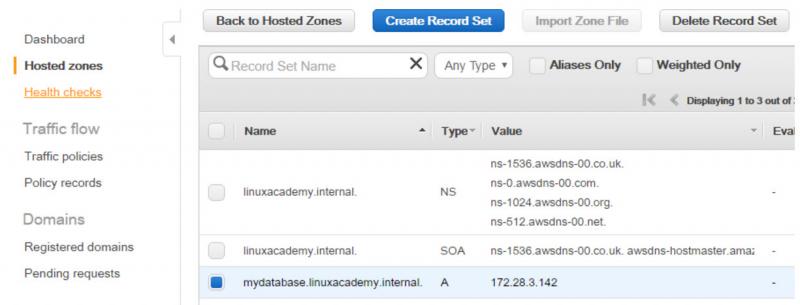
|  |
| --- |
| resource "aws\_subnet" "PublicAZA" {  vpc\_id = "${aws\_vpc.terraformmain.id}"  cidr\_block = "${var.Subnet-Public-AzA-CIDR}"  tags {  Name = "PublicAZA"  }  availability\_zone = "${data.aws\_availability\_zones.available.names[0]}"  }  resource "aws\_route\_table\_association" "PublicAZA" {  subnet\_id = "${aws\_subnet.PublicAZA.id}"  route\_table\_id = "${aws\_route\_table.public.id}"  }  resource "aws\_subnet" "PrivateAZA" {  vpc\_id = "${aws\_vpc.terraformmain.id}"  cidr\_block = "${var.Subnet-Private-AzA-CIDR}"  tags {  Name = "PublicAZB"  }  availability\_zone = "${data.aws\_availability\_zones.available.names[1]}"  }  resource "aws\_route\_table\_association" "PrivateAZA" {  subnet\_id = "${aws\_subnet.PrivateAZA.id}"  route\_table\_id = "${aws\_route\_table.private.id}"  } |

There are two subnets associated with the respective routes: a public and a private.

**dns-and-dhcp.tf**

|  |
| --- |
| resource "aws\_vpc\_dhcp\_options" "mydhcp" {  domain\_name = "${var.DnsZoneName}"  domain\_name\_servers = ["AmazonProvidedDNS"]  tags {  Name = "My internal name"  }  }  resource "aws\_vpc\_dhcp\_options\_association" "dns\_resolver" {  vpc\_id = "${aws\_vpc.terraformmain.id}"  dhcp\_options\_id = "${aws\_vpc\_dhcp\_options.mydhcp.id}"  }  /\* DNS PART ZONE AND RECORDS \*/  resource "aws\_route53\_zone" "main" {  name = "${var.DnsZoneName}"  vpc\_id = "${aws\_vpc.terraformmain.id}"  comment = "Managed by terraform"  }  resource "aws\_route53\_record" "database" {  zone\_id = "${aws\_route53\_zone.main.zone\_id}"  name = "mydatabase.${var.DnsZoneName}"  type = "A"  ttl = "300"  records = ["${aws\_instance.database.private\_ip}"]  } |

In this file, three things were accomplished: the private Route53 DNS zone was created, the association with the VPC was made, and the DNS record for the database was created. Terraform perform the actions in the right order, the last component in this file will be the database dns record because it depends on the private ip of the EC2 database machine. This machine will be allocated during the database creation.



**securitygroups.tf**

|  |
| --- |
| resource "aws\_security\_group" "FrontEnd" {  name = "FrontEnd"  tags {  Name = "FrontEnd"  }  description = "ONLY HTTP CONNECTION INBOUD"  vpc\_id = "${aws\_vpc.terraformmain.id}"  ingress {  from\_port = 80  to\_port = 80  protocol = "TCP"  cidr\_blocks = ["0.0.0.0/0"]  }  ingress {  from\_port = "22"  to\_port = "22"  protocol = "TCP"  cidr\_blocks = ["0.0.0.0/0"]  }  egress {  from\_port = 0  to\_port = 0  protocol = "-1"  cidr\_blocks = ["0.0.0.0/0"]  }  }  resource "aws\_security\_group" "Database" {  name = "Database"  tags {  Name = "Database"  }  description = "ONLY tcp CONNECTION INBOUND"  vpc\_id = "${aws\_vpc.terraformmain.id}"  ingress {  from\_port = 3306  to\_port = 3306  protocol = "TCP"  security\_groups = ["${aws\_security\_group.FrontEnd.id}"]  }  ingress {  from\_port = "22"  to\_port = "22"  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"]  }  } |

We have two security groups: one for the web application, and another for the database. They both need to have the outbound (egress) rule to have internet access because yum will install the Apache and MySQL servers, but the connection to the MySQL port will be allowed only from instances that belong to the webapp security group.

I have left the ssh port open only for debug reason, but you can also delete that rule.

**ec2-machines.tf**

|  |
| --- |
| resource "aws\_instance" "phpapp" {  ami = "${lookup(var.AmiLinux, var.region)}"  instance\_type = "t2.micro"  associate\_public\_ip\_address = "true"  subnet\_id = "${aws\_subnet.PublicAZA.id}"  vpc\_security\_group\_ids = ["${aws\_security\_group.FrontEnd.id}"]  key\_name = "${var.key\_name}"  tags {  Name = "phpapp"  }  user\_data = <<HEREDOC  #!/bin/bash  yum update -y  yum install -y httpd24 php56 php56-mysqlnd  service httpd start  chkconfig httpd on  echo "<?php" >> /var/www/html/calldb.php  echo "\$conn = new mysqli('mydatabase.saifulcce.internal', 'root', 'secret', 'test');" >> /var/www/html/calldb.php  echo "\$sql = 'SELECT \* FROM mytable'; " >> /var/www/html/calldb.php  echo "\$result = \$conn->query(\$sql); " >> /var/www/html/calldb.php  echo "while(\$row = \$result->fetch\_assoc()) { echo 'the value is: ' . \$row['mycol'] ;} " >> /var/www/html/calldb.php  echo "\$conn->close(); " >> /var/www/html/calldb.php  echo "?>" >> /var/www/html/calldb.php  HEREDOC  }  resource "aws\_instance" "database" {  ami = "${lookup(var.AmiLinux, var.region)}"  instance\_type = "t2.micro"  associate\_public\_ip\_address = "false"  subnet\_id = "${aws\_subnet.PrivateAZA.id}"  vpc\_security\_group\_ids = ["${aws\_security\_group.Database.id}"]  key\_name = "${var.key\_name}"  tags {  Name = "database"  }  user\_data = <<HEREDOC  #!/bin/bash  yum update -y  yum install -y mysql55-server  service mysqld start  /usr/bin/mysqladmin -u root password 'secret'  mysql -u root -psecret -e "create user 'root'@'%' identified by 'secret';" mysql  mysql -u root -psecret -e 'CREATE TABLE mytable (mycol varchar(255));' test  mysql -u root -psecret -e "INSERT INTO mytable (mycol) values ('saifulccethebest') ;" test  HEREDOC  } |

We chose an AWS Linux AMI. I loaded the userdata using the HEREDOC option, but you can also use an external file.

**The database machine**

This machine is placed in the private subnet and has its security group. The userdata performs the following actions:

* update the OS
* install the MySQL server and run it
* configure the root user to grant access from other machines
* create a table in the test database and add one line inside

**The webapp machine**

It is placed in the public subnet so it is possible to reach it from your browser using port 80. The userdata performs the following actions:

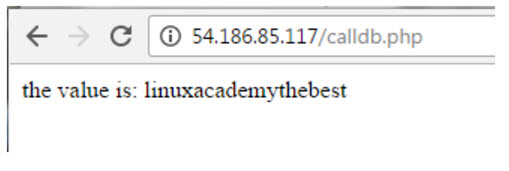
* update the OS
* install the Apache web server and its php module
* start the Apache
* using the echo command place in the public directory, a php file that reads the value inside the database created in the other EC2

**Running the terraform and connect to the application**

Create all files with extension .tf inside a directory, replace the values in the variable.tf as explained in the first part of the article, and then run the command:

**terraform apply**

After a few minutes, the process should be completed and you can go to your AWS web console and read the public ip of your EC2 machine. Visit the url in your browser, and you will see the result of the php command.



Testing the zone

To test your internal DNS routing system, you can log in inside the web server machine to run a DNS query for the private zone like this:

$ host mydatabase.saifulcce.internal

mydatabase.saifulcce.internal has address 172.28.3.142

If you try to do it from a machine outside the vpc, you will have:

host mydatabase.saifulcce.internal.

Host mydatabase.saifulcce.internal. not found: 3(NXDOMAIN)