- OpenStack

- Big Data

Popular Certification Courses

AWS AWS Certified Cloud Practitioner Certified Solutions Architect - Associate Certified Developer - Associate Certified SysOps Administrator - Associate Certified DevOps Engineer - Professional Certified Solutions Architect - Professional Certified Big Data Specialty Certified Advanced Networking Specialty
Google Cloud Platform Google Certified Professional - Cloud Architect - Part 1 Google Certified Professional - Cloud Architect - Part 2 Google Certified Professional - Cloud Architect - Part 3
Red Hat Linux Academy Red Hat Certified Systems Administrator - RHCSA (EX200) Prep Course Linux Academy Red Hat Certified Engineer - RHCE (EX300) Prep Course Linux Academy Red Hat Certified Engineer in Red Hat OpenStack (EX310K) Prep Course Linux Academy Red Hat Certified Specialist in Containerized Application Development (EX276) Prep Course Linux Academy Red Hat Certified Specialist in Ansible Automation (EX407) Prep Course Linux Academy Red Hat Certified Specialist in Server Hardening (EX413) Prep Course (OpenShift) Linux Academy Red Hat Certified Specialist in Platform-as-a-Service exam (EX280) Prep Course
Linux Professional Institute (LPI) Linux Essentials - Exam 010 - Newly Refreshed! Linux+ and LPIC-1: System Administrator - Exam 101 Linux+ and LPIC-1: System Administrator - Exam 102 LPIC-2: Linux Engineer Exam 201 LPIC-2: Linux Engineer Exam 202
Microsoft Developing Microsoft Azure Solutions Exam 70-532 Implementing Microsoft Azure Infrastructure Exam 70-533 Architecting Microsoft Azure Solutions Exam 70-534
Containers Cloud Native (CKA) Certified Kubernetes Administrator Docker Certified Associate Prep Course
Linux Foundation Certified System Administrator Certified Systems Engineer
OpenStack Linux Academy Red Hat Certified Engineer in Red Hat OpenStack Prep Course (EX310K) Linux Academy's OpenStack Certified Mirantis 100 Prep (OCM100) OpenStack Foundation Certified OpenStack Administrator
CompTIA Linux+ LX0-103 Linux+ LX0-104 CompTIA Cloud Essentials Certification
DevOps Certified Chef Developer Basic Chef Fluency Badge Docker Certified Associate Prep Course Chef Local Cookbook Development Badge Certified Jenkins Engineer (CJE) Nagios Certified Professional PPT-204: Puppet 204 - System Administration Using Puppet

Back to Guides

Sign up for free and access the community!

Share:

A complete AWS environment with Terraform

Posted on: Jan 18, 2017 by: Giuseppe B

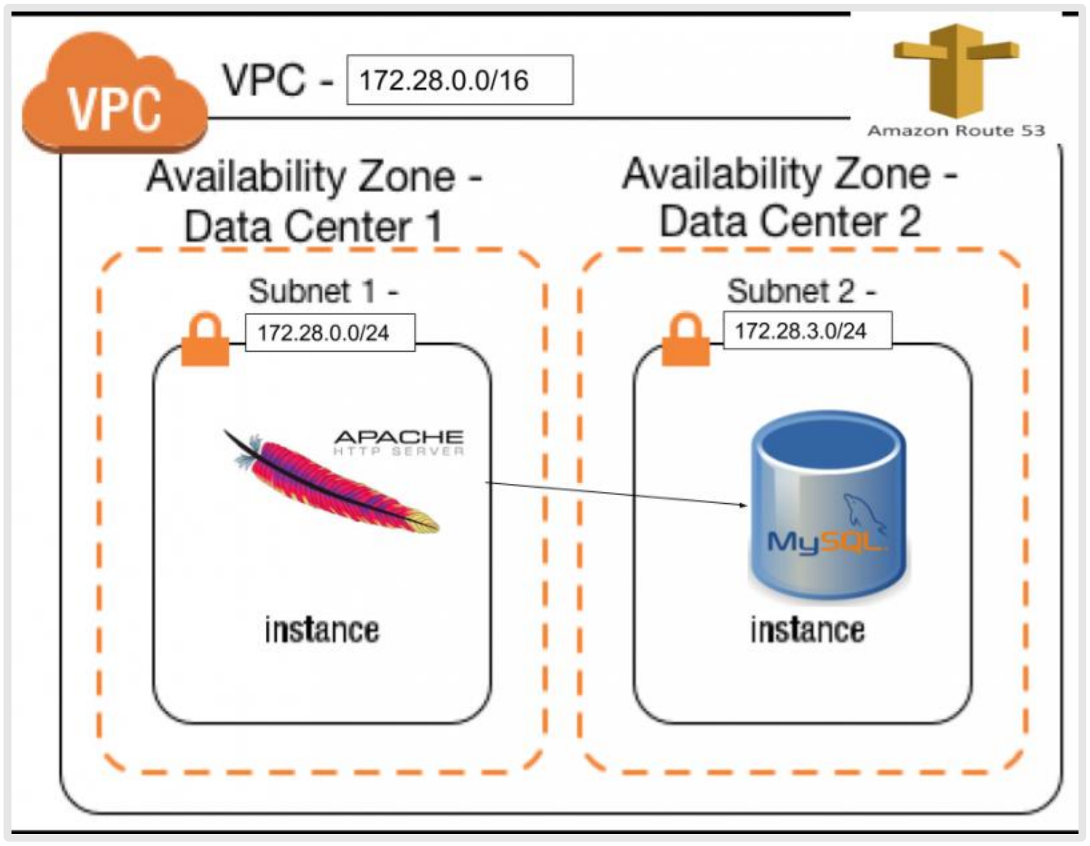
Table of Contents

Related Guides

1. A complete AWS environment with Terraform
2. Automating Terraform with Jenkins and AWS CodeCommit

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. Having Terraform installed: it is pretty easy to install it if you haven't already. You can find the instructions in my first article "Introduction to Terraform Modules."
2. 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. See how to create a pem key here if you haven't already.

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 = "linuxacademy.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

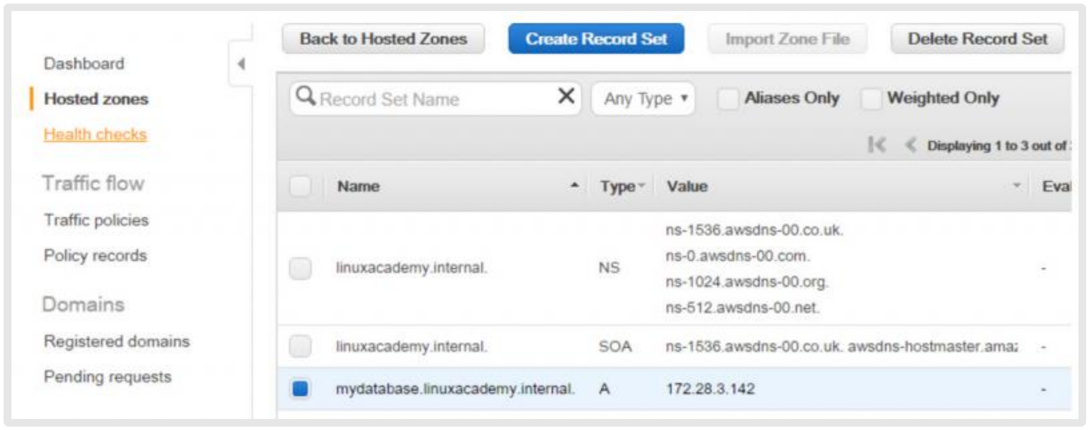
```
resource "aws_vpc_dhcp_options" "mydhcp" {
    domain_name = "${var.DnsZoneName}"
    domain_name_servers = ["AmazonProvidedDNS"]
    tags {
      Name = "My internal name"
    }
}

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.



Name	Type	Value	Eval
linuxacademy.internal.	NS	ns-1536.awsdns-00.co.uk. ns-0.awsdns-00.com. ns-1024.awsdns-00.org. ns-512.awsdns-00.net.	-
linuxacademy.internal.	SOA	ns-1536.awsdns-00.co.uk. awsdns-hostmaster.ama:	-
mydatabase.linuxacademy.internal.	A	172.28.3.142	-

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.linuxacademy.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 ('linuxacademythebest') ;" 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:

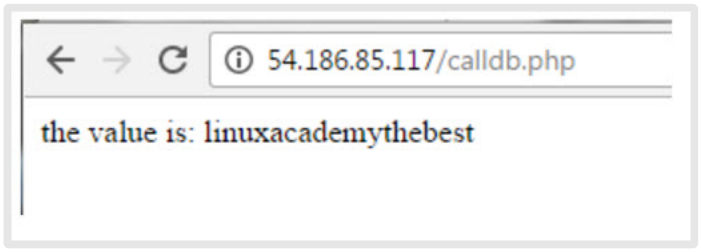
- update the OS
- install the Apache web server and its php module
- start the Apache
- using the echo command place in the public directory, a php file that reads the value inside the database created in the other EC2

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.


54.186.85.117/calldb.php

the value is: linuxacademythebest

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.linuxacademy.internal
mydatabase.linuxacademy.internal has address 172.28.3.142
```

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

```
host mydatabase.linuxacademy.internal.
Host mydatabase.linuxacademy.internal. not found: 3(NXDOMAIN)
```

YouTube Tutorial and github repo

There is also an YouTube Tutorial and a github repo to download the files.

-
  Davis E
  01-18-2017

  Nice work on the guide! It looks good.

-
  Davis E
  01-18-2017

  I fixed some formatting and table-of-content issues. Let me know if you notice something I accidentally messed up! @pippopeppe83

-
  Akshay T
  02-07-2017

  i m unable to download the zip file using wget in the terminal during lab, hence not able to proceed, can you please help, the file is getting downloaded, but unable to unzip that as that is a corrupted file

-
  Giuseppe B
  02-08-2017