Hosting a simple website on AWS

Code Documentation

In this project, we'll be hosting a highly available, and scalable website on Amazon web services (AWS) cloud using the infrastructure-as-code language, Terraform.

The website will be hosted on Apache servers running within the multiple EC2 instances, managed by auto scaling groups, over multiple availability zones. The incoming traffic will be distributed among them with a load balancer.

Getting started

To deploy the web page, please make sure you have an AWS account, and the HashiCorp terraform installed on your computer.

You can download the Terraform from the official source, https://www.terraform.io/

Setting up the AWS credentials

First, It is important to set the AWS credentials,

There are two ways to do this,

Method 1: Setting the environment variables by running the commands, (preferably within a virtual environment of the project)

```
export AWS_ACCESS_KEY_ID="your_access_key_here"
export AWS_SECRET_ACCESS_KEY="your_secret_key_here"
```

Method 2: Include your access key and secret key directly within the terraform code (Not recommended due to security reasons)

```
# Configuring AWS Provider

provider "aws" {
region = "us-east-1"
# Set environment variables or enter the keys here
# access_key = "your_access_key_here"
# secret_key = "your_secret_key_here"
}
```

Initializing the terraform

To Initialize terraform, please run the following command,

```
terraform init
```

Deploying the web page

Generate the plan by running,

```
terraform plan
```

Deploy the webpage on the cloud by running,

```
terraform apply
```

You can get the link to the website as an output.

Terminating the network

You can shut down the webpage and terminate the running instances, simply run,

```
terraform destroy
```

Code Documentation

VPC

AWS VPC is a secure, virtual private cloud that resembles a traditional network that you operate on an architecture. Creation of VPC allows us to launch our subnets within the network.

```
# creating a VPC

resource "aws_vpc" "vpc" {
cidr_block = "172.16.0.0/16"

tags = {
  Name = "vpc"
}
}
```

We allocate the class B, IP- address 172.16.0.0/16 to our VPC.

Internet gateway

```
# Creating an Internet gateway

resource "aws_internet_gateway" "internet_gateway" {
vpc_id = aws_vpc.vpc.id

tags = {
  Name = "internet-gateway"
}
}
```

Internet gateway is a horizontally scaled network component that allows our VPC to access the internet, which is essentially the internet gateway of our network.

Subnets

We create several private and public subnets within the given availability zones for hosting the EC2 instances on the zones.

```
# Creating public subnet

resource "aws_subnet" "public_subnets" {
count                   = length(var.public_subnet_cidr)
vpc_id                  = aws_vpc.vpc.id
cidr_block              = var.public_subnet_cidr[count.index]
map_public_ip_on_launch = true
availability_zone       = var.az_names[count.index]

tags = {
  Name = join("-", ["public-subnet", var.az_names[count.index]])
}
}

# Creating private subnet

resource "aws_subnet" "private_subnets" {
count             = length(var.private_subnet_cidr)
vpc_id            = aws_vpc.vpc.id
cidr_block        = var.private_subnet_cidr[count.index]
availability_zone = var.az_names[count.index]

tags = {
  Name = join("-", ["private-subnet", var.az_names[count.index]])
}
}
```

Route tables

```
# Creating public route table
```

```
resource "aws_route_table" "public_route_table" {
vpc_id = aws_vpc.vpc.id

route {
  cidr_block = "0.0.0.0/0"
  gateway_id = aws_internet_gateway.internet_gateway.id
}

  tags = {
  Name = "public-route-table"
}
}

# Creating private route table

resource "aws_route_table" "private_route_table" {

count = length(var.private_subnet_cidr)
vpc_id = aws_vpc.vpc.id

tags = {
  Name = join("-", ["private_route_table", var.az_names[count.index]])
}
}
```

We create a public route table that allows the NAT gateway to access the internet gateway id.

Elastic IP

```
# Creating elastic ip

resource "aws_eip" "elastic_ip" {
count = length(var.private_subnet_cidr)
domain = "vpc"
tags = {
  Name = join("-", ["elastic-ip", var.az_names[count.index]])
}
}
```

We create elastic IP for the subnets within our VPC.

NAT Gateway

```
# Creating NAT gateway

resource "aws_nat_gateway" "nat_gateway" {
count = length(var.public_subnet_cidr)
connectivity_type = "public"
allocation_id     = aws_eip.elastic_ip[count.index].id
subnet_id         = aws_subnet.public_subnets[count.index].id
depends_on = [aws_internet_gateway.internet_gateway]

tags = {
  Name = join("-", ["nat-gateway", var.az_names[count.index]])
}
```

We operate Network address Translation service (NAT) gateways within our public subnets so that we can allow our EC2 instances to access the internet gateways.

Route

```
# Route

resource "aws_route" "private-to-nat"{

count = length(var.private_subnet_cidr)
destination_cidr_block = "0.0.0.0/0"
route_table_id = aws_route_table.private_route_table[count.index].id
nat_gateway_id = aws_nat_gateway.nat_gateway[count.index].id
}

# Creating public route table association

resource "aws_route_table_association" "public_route_association" {
count          = length(var.public_subnet_cidr)
subnet_id      = aws_subnet.public_subnets[count.index].id
route_table_id = aws_route_table.public_route_table.id
}

# creating private route table association

resource "aws_route_table_association" "private_route_association" {
count          = length(var.private_subnet_cidr)
subnet_id      = aws_subnet.private_subnets[count.index].id
route_table_id = aws_route_table.private_route_table[count.index].id
}
```

Security Groups

Security Groups acts as virtual firewall for our network that allows traffic only to the authorized ports.

```
# Security Group Resources

resource "aws_security_group" "security_group_lb" {
name        = "security-group-lb"
description = "Security Group for load balancer"
vpc_id      = aws_vpc.vpc.id

ingress {
  description = "HTTP from Internet"
  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"]
}

tags = {
  Name = "security-group-load-balancer"
}
}
```

Launch Template for EC2 instances

The number of EC2 instances will be launched based on the launch template configurations. for our project, we'll be using the Amazon-Linux OS image for the EC2 instance. We configure the instance type as 't2.micro', which is a burstable compute service.

```
# Launch Template for EC2 instances
```

```
resource "aws_launch_template" "launch_template" {
name          = "ec2-launch-template"
image_id      = "ami-08a0d1e16fc3f61ea"
instance_type = "t2.micro"

network_interfaces {
  device_index    = 0
  security_groups = [aws_security_group.asg_security_group.id]
}
tag_specifications {
  resource_type = "instance"

  tags = {
  Name = "ec2-launch-template"
  }
}

user_data = filebase64("webserver.sh")
}
```

We finally run the 'webserver.sh' bash file for installing, initializing and configuring the Apache httpd webserver. Finally, we place our webpage as 'index.html' page within /var/www/html/ folder of the EC2 instance, the default document root of Apache server.

Auto Scaling Group

```
# Auto Scaling Group

resource "aws_autoscaling_group" "auto_scaling_group" {
desired_capacity    = 3
max_size            = 4
min_size            = 3
vpc_zone_identifier = [for i in aws_subnet.private_subnets[*] : i.id]
target_group_arns   = [aws_lb_target_group.lb_target_group.arn]

launch_template {
  id      = aws_launch_template.launch_template.id
  version = aws_launch_template.launch_template.latest_version
}
}
```

The auto scaling group launches the provided launch templates within our subnets to match the incoming traffic or computational needs. For this project, we restrict the minimum capacity to 3 and maximum capacity to 4.

Load Balancer

```
# Load Balancer

resource "aws_lb" "load_balancer" {
name               = "load-balancer"
internal           = false
load_balancer_type = "application"
security_groups    = [aws_security_group.security_group_lb.id]
subnets            = [for i in aws_subnet.public_subnets : i.id]
tags = {
  Name = "Load-balancer"
}
}
```

The load balancer manages the incoming traffic and sends them to available EC2 instances. This would help us manage the load on our servers.

Load Balancer target group

```
# Load Balancer target group

resource "aws_lb_target_group" "lb_target_group" {
name     = "lb-target-group"
port     = 80
protocol = "HTTP"
vpc_id   = aws_vpc.vpc.id

health_check {
  path    = "/"
  matcher = 200
}
}
```

The Load balancer target groups sets the target to HTTP at port 80. It also judges the EC2 instance health based on the HTTP 200 success response of the server.

Load Balancer listener

Load balancer listen to the available EC2 instances, and gauges their health.

```
# Load Balancer listener

resource "aws_lb_listener" "lb_listener" {
load_balancer_arn = aws_lb.load_balancer.arn
port              = "80"
protocol          = "HTTP"

default_action {
  type             = "forward"
  target_group_arn = aws_lb_target_group.lb_target_group.arn
}
}
```

Output: Load balancer url

```
# Output Load balancer url

output "load_balancer_url" {
description = "link of the webpage"
value       = "http://${aws_lb.load_balancer.dns_name}"
}
```

At the end of the execution of terraform apply command, you will get the http link of the website hosted on the AWS cloud.