**Projects**

* [Projects](https://darey.io/project-listings/)
* [Docs](https://www.darey.io/projects/)
* [(STEP 29) Project 17 Automate Infrastructure With IAC using Terraform Part 2](https://www.darey.io/docs/automate-infrastructure-with-iac-using-terraform-part-2/)

**AUTOMATE INFRASTRUCTURE WITH IAC USING TERRAFORM. PART 2**

[Start Project](https://www.darey.io/docs/automate-infrastructure-with-iac-using-terraform-part-2/?checkread=1)Submit Project for Review

Before we go deeper into automating other parts of our infrastructure on AWS, it is very important to fully understand certain concepts around **Networking** (in case this is completely new area to you). Networking is a very broad topic and some of internals of Terraform modules related to Networking, like cidrsubnet(), may still not be fully clear to you.

To fully clear your understanding, we highly recommend you watching Networking videos by **Eli the Computer Guy** on YouTube, and in addition to that, read following golden articles on **Networking Terminology, Interfaces, Protocols, IP Address, Subnets, and CIDR Notation** by **Justin Ellingwood** from **Digital Ocean**.

Eli the Computer Guy videos

* [Introduction to Networking](https://youtu.be/rL8RSFQG8do)
* [TCP/IP and Subnet Masking](https://youtu.be/EkNq4TrHP_U)

If you are interested to dive deeper into Networking domain, you can watch the entire playlist [here](https://www.youtube.com/playlist?list=PLF360ED1082F6F2A5)

Justin Ellingwood blog posts

**WARNING**: You may initially feel overwhelmed by the information provided in these articles. It is fine if you don not fully understand them the first time. Bookmark the pages, read them again and again for the next few days. Find other articles talking about the same topics on Google and watch YouTube videos about them. Subconsciously, you will begin to understand them over time.

* [Networking Part 1](https://www.digitalocean.com/community/tutorials/an-introduction-to-networking-terminology-interfaces-and-protocols)
* [Networking Part 1](https://www.digitalocean.com/community/tutorials/understanding-ip-addresses-subnets-and-cidr-notation-for-networking#netmasks-and-subnets)

Instructions On How To Submit Your Work For Review And Feedback

To submit your work for review and feedback – follow [**this instruction**](https://starter-pbl.darey.io/en/latest/submission.html).

Continue Infrastructure Automation with Terraform

Let us continue from where we have stopped in [Project 16](https://www.darey.io/docs/project-16-introduction/).

Based on the knowledge from the previous project lets keep on creating AWS resources!

**Networking**

**Private subnets & best practices**

Create 4 private subnets keeping in mind following principles:

* Make sure you use variables or length() function to determine the number of AZs
* Use variables and cidrsubnet() function to allocate vpc\_cidr for subnets
* Keep variables and resources in separate files for better code structure and readability
* Tags all the resources you have created so far. Explore how to use format() and count functions to automatically tag subnets with its respective number.

A little bit more about Tagging

Tagging is a straightforward, but a very powerful concept that helps you manage your resources much more efficiently:

* Resources are much better organized in ‘virtual’ groups
* They can be easily filtered and searched from console or programmatically
* Billing team can easily generate reports and determine how much each part of infrastructure costs how much (by department, by type, by environment, etc.)
* You can easily determine resources that are not being used and take actions accordingly
* If there are different teams in the organisation using the same account, tagging can help differentiate who owns which resources.

**Note:** You can add multiple tags as a default set. for example, in out terraform.tfvars file we can have default tags defined.

tags = {

Enviroment = "production"

Owner-Email = "dare@darey.io"

Managed-By = "Terraform"

Billing-Account = "1234567890"

}

Now you can tag all you resources using the format below

tags = merge(

var.tags,

{

Name = "Name of the resource"

},

)

**NOTE:** Update the variables.tf to declare the variable tags used in the format above;

variable "tags" {

description = "A mapping of tags to assign to all resources."

type = map(string)

default = {}

}

The nice thing about this is – anytime we need to make a change to the tags, we simply do that in one single place (terraform.tfvars).

But, our key-value pairs are hard coded. So, go ahead and work out a fix for that. Simply create variables for each value and use var.variable\_name as the value to each of the keys.  
Apply the same best practices for all other resources you will create further.

Internet Gateways & format() function

Create an Internet Gateway in a separate Terraform file internet\_gateway.tf

resource "aws\_internet\_gateway" "ig" {

vpc\_id = aws\_vpc.main.id

tags = merge(

var.tags,

{

Name = format("%s-%s!", aws\_vpc.main.id,"IG")

}

)

}

Did you notice how we have used format() function to dynamically generate a unique name for this resource? The first part of the %s takes the interpolated value of aws\_vpc.main.id while the second %s appends a literal string IG and finally an exclamation mark is added in the end.

If any of the resources being created is either using the count function, or creating multiple resources using a loop, then a key-value pair that needs to be unique must be handled differently.

For example, each of our subnets should have a unique name in the tag section. Without the format() function, we would not be able to see uniqueness. With the format function, each private subnet’s tag will look like this.

Name = PrvateSubnet-0

Name = PrvateSubnet-1

Name = PrvateSubnet-2

Lets try and see that in action.

tags = merge(

var.tags,

{

Name = format("PrivateSubnet-%s", count.index)

}

)

NAT Gateways

Create 1 NAT Gateways and 1 Elastic IP (EIP) addresses

Now use similar approach to create the NAT Gateways in a new file called natgateway.tf.

**Note:** We need to create an Elastic IP for the NAT Gateway, and you can see the use of depends\_on to indicate that the Internet Gateway resource must be available before this should be created. Although Terraform does a good job to manage dependencies, but in some cases, it is good to be explicit.

You can read more on dependencies [here](https://www.terraform.io/docs/language/meta-arguments/depends_on.html)

resource "aws\_eip" "nat\_eip" {

vpc = true

depends\_on = [aws\_internet\_gateway.ig]

tags = merge(

var.tags,

{

Name = format("%s-EIP", var.name)

},

)

}

resource "aws\_nat\_gateway" "nat" {

allocation\_id = aws\_eip.nat\_eip.id

subnet\_id = element(aws\_subnet.public.\*.id, 0)

depends\_on = [aws\_internet\_gateway.ig]

tags = merge(

var.tags,

{

Name = format("%s-Nat", var.name)

},

)

}

# AWS ROUTES

Create a file called route\_tables.tf and use it to create routes for both public and private subnets, create the below resources. Ensure they are properly tagged.

* aws\_route\_table
* aws\_route
* aws\_route\_table\_association

# create private route table

resource "aws\_route\_table" "private-rtb" {

vpc\_id = aws\_vpc.main.id

tags = merge(

var.tags,

{

Name = format("%s-Private-Route-Table", var.name)

},

)

}

# associate all private subnets to the private route table

resource "aws\_route\_table\_association" "private-subnets-assoc" {

count = length(aws\_subnet.private[\*].id)

subnet\_id = element(aws\_subnet.private[\*].id, count.index)

route\_table\_id = aws\_route\_table.private-rtb.id

}

# create route table for the public subnets

resource "aws\_route\_table" "public-rtb" {

vpc\_id = aws\_vpc.main.id

tags = merge(

var.tags,

{

Name = format("%s-Public-Route-Table", var.name)

},

)

}

# create route for the public route table and attach the internet gateway

resource "aws\_route" "public-rtb-route" {

route\_table\_id = aws\_route\_table.public-rtb.id

destination\_cidr\_block = "0.0.0.0/0"

gateway\_id = aws\_internet\_gateway.ig.id

}

# associate all public subnets to the public route table

resource "aws\_route\_table\_association" "public-subnets-assoc" {

count = length(aws\_subnet.public[\*].id)

subnet\_id = element(aws\_subnet.public[\*].id, count.index)

route\_table\_id = aws\_route\_table.public-rtb.id

}

Now if you run terraform plan and terraform apply it will add the following resources to AWS in multi-az set up:

* – Our main vpc
* – 2 Public subnets
* – 4 Private subnets
* – 1 Internet Gateway
* – 1 NAT Gateway
* – 1 EIP
* – 2 Route tables

Now, we are done with Networking part of AWS set up, let us move on to Compute and Access Control configuration automation using Terraform!

### AWS Identity and Access Management

#### [IaM](https://docs.aws.amazon.com/iam/index.html) and [Roles](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles.html)

We want to pass an IAM role our EC2 instances to give them access to some specific resources, so we need to do the following:

1. Create [AssumeRole](https://docs.aws.amazon.com/STS/latest/APIReference/API_AssumeRole.html)

Assume Role uses Security Token Service (STS) API that returns a set of temporary security credentials that you can use to access AWS resources that you might not normally have access to. These temporary credentials consist of an access key ID, a secret access key, and a security token. Typically, you use AssumeRole within your account or for cross-account access.

Add the following code to a new file named roles.tf

resource "aws\_iam\_role" "ec2\_instance\_role" {

name = "ec2\_instance\_role"

assume\_role\_policy = jsonencode({

Version = "2012-10-17"

Statement = [

{

Action = "sts:AssumeRole"

Effect = "Allow"

Sid = ""

Principal = {

Service = "ec2.amazonaws.com"

}

},

]

})

tags = merge(

var.tags,

{

Name = "aws assume role"

},

)

}

In this code we are creating AssumeRole with AssumeRole policy. It grants to an entity, in our case it is an EC2, permissions to assume the role.

1. Create [IAM policy](https://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_create.html) for this role

This is where we need to define a required policy (i.e., permissions) according to our requirements. For example, allowing an IAM role to perform action describe applied to EC2 instances:

resource "aws\_iam\_policy" "policy" {

name = "ec2\_instance\_policy"

description = "A test policy"

policy = jsonencode({

Version = "2012-10-17"

Statement = [

{

Action = [

"ec2:Describe\*",

]

Effect = "Allow"

Resource = "\*"

},

]

})

tags = merge(

var.tags,

{

Name = "aws assume policy"

},

)

}

1. Attach the Policy to the IAM Role

This is where, we will be attaching the policy which we created above, to the role we created in the first step.

resource "aws\_iam\_role\_policy\_attachment" "test-attach" {

role = aws\_iam\_role.ec2\_instance\_role.name

policy\_arn = aws\_iam\_policy.policy.arn

}

1. Create an [Instance Profile](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2_instance-profiles.html) and interpolate the IAM Role

resource "aws\_iam\_instance\_profile" "ip" {

name = "aws\_instance\_profile\_test"

role = aws\_iam\_role.ec2\_instance\_role.name

}

We are pretty much done with Identity and Management part for now, let us move on and create other resources required.

#### Resources to be created

As per our architecture we need to do the following:

1. Create Security Groups
2. Create Target Group for Nginx, WordPress and Tooling
3. Create certificate from AWS certificate manager
4. Create an External Application Load Balancer and Internal Application Load Balancer.
5. create launch template for Bastion, Tooling, Nginx and WordPress
6. Create an Auto Scaling Group (ASG) for Bastion, Tooling, Nginx and WordPress
7. Create Elastic Filesystem
8. Create Relational Database (RDS)

Let us create some Terraform configuration code to accomplish these tasks.

# CREATE SECURITY GROUPS

We are going to create all the security groups in a single file, then we are going to refrence this security group within each resources that needs it.

**IMPORTANT:**

* Check out the terraform documentation for [security group](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/security_group)
* Check out the terraform documentation for [security group rule](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/security_group_rule)

Create a file and name it security.tf, copy and paste the code below

# security group for alb, to allow acess from any where for HTTP and HTTPS traffic

resource "aws\_security\_group" "ext-alb-sg" {

name = "ext-alb-sg"

vpc\_id = aws\_vpc.main.id

description = "Allow TLS inbound traffic"

ingress {

description = "HTTP"

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = ["0.0.0.0/0"]

}

ingress {

description = "HTTPS"

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"]

}

tags = merge(

var.tags,

{

Name = "ext-alb-sg"

},

)

}

# security group for bastion, to allow access into the bastion host from you IP

resource "aws\_security\_group" "bastion\_sg" {

name = "vpc\_web\_sg"

vpc\_id = aws\_vpc.main.id

description = "Allow incoming HTTP connections."

ingress {

description = "SSH"

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"]

}

tags = merge(

var.tags,

{

Name = "Bastion-SG"

},

)

}

#security group for nginx reverse proxy, to allow access only from the extaernal load balancer and bastion instance

resource "aws\_security\_group" "nginx-sg" {

name = "nginx-sg"

vpc\_id = aws\_vpc.main.id

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

tags = merge(

var.tags,

{

Name = "nginx-SG"

},

)

}

resource "aws\_security\_group\_rule" "inbound-nginx-http" {

type = "ingress"

from\_port = 443

to\_port = 443

protocol = "tcp"

source\_security\_group\_id = aws\_security\_group.ext-alb-sg.id

security\_group\_id = aws\_security\_group.nginx-sg.id

}

resource "aws\_security\_group\_rule" "inbound-bastion-ssh" {

type = "ingress"

from\_port = 22

to\_port = 22

protocol = "tcp"

source\_security\_group\_id = aws\_security\_group.bastion\_sg.id

security\_group\_id = aws\_security\_group.nginx-sg.id

}

# security group for ialb, to have acces only from nginx reverser proxy server

resource "aws\_security\_group" "int-alb-sg" {

name = "my-alb-sg"

vpc\_id = aws\_vpc.main.id

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

tags = merge(

var.tags,

{

Name = "int-alb-sg"

},

)

}

resource "aws\_security\_group\_rule" "inbound-ialb-https" {

type = "ingress"

from\_port = 443

to\_port = 443

protocol = "tcp"

source\_security\_group\_id = aws\_security\_group.nginx-sg.id

security\_group\_id = aws\_security\_group.int-alb-sg.id

}

# security group for webservers, to have access only from the internal load balancer and bastion instance

resource "aws\_security\_group" "webserver-sg" {

name = "my-asg-sg"

vpc\_id = aws\_vpc.main.id

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

tags = merge(

var.tags,

{

Name = "webserver-sg"

},

)

}

resource "aws\_security\_group\_rule" "inbound-web-https" {

type = "ingress"

from\_port = 443

to\_port = 443

protocol = "tcp"

source\_security\_group\_id = aws\_security\_group.int-alb-sg.id

security\_group\_id = aws\_security\_group.webserver-sg.id

}

resource "aws\_security\_group\_rule" "inbound-web-ssh" {

type = "ingress"

from\_port = 22

to\_port = 22

protocol = "tcp"

source\_security\_group\_id = aws\_security\_group.bastion\_sg.id

security\_group\_id = aws\_security\_group.webserver-sg.id

}

# security group for datalayer to alow traffic from websever on nfs and mysql port and bastiopn host on mysql port

resource "aws\_security\_group" "datalayer-sg" {

name = "datalayer-sg"

vpc\_id = aws\_vpc.main.id

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

tags = merge(

var.tags,

{

Name = "datalayer-sg"

},

)

}

resource "aws\_security\_group\_rule" "inbound-nfs-port" {

type = "ingress"

from\_port = 2049

to\_port = 2049

protocol = "tcp"

source\_security\_group\_id = aws\_security\_group.webserver-sg.id

security\_group\_id = aws\_security\_group.datalayer-sg.id

}

resource "aws\_security\_group\_rule" "inbound-mysql-bastion" {

type = "ingress"

from\_port = 3306

to\_port = 3306

protocol = "tcp"

source\_security\_group\_id = aws\_security\_group.bastion\_sg.id

security\_group\_id = aws\_security\_group.datalayer-sg.id

}

resource "aws\_security\_group\_rule" "inbound-mysql-webserver" {

type = "ingress"

from\_port = 3306

to\_port = 3306

protocol = "tcp"

source\_security\_group\_id = aws\_security\_group.webserver-sg.id

security\_group\_id = aws\_security\_group.datalayer-sg.id

}

**IMPORTANT NOTE:** We used the aws\_security\_group\_rule to refrence another security group in a security group.

# CREATE CERTIFICATE FROM AMAZON CERIFICATE MANAGER

Create cert.tf file and add the following code snippets to it.

**NOTE:** Read Through to change the domain name to your own domain name and every other name that needs to be changed.

* Check out the terraform documentation for [AWS Certifivate mangarer](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/acm_certificate)

# The entire section create a certiface, public zone, and validate the certificate using DNS method

# Create the certificate using a wildcard for all the domains created in oyindamola.gq

resource "aws\_acm\_certificate" "oyindamola" {

domain\_name = "\*.oyindamola.gq"

validation\_method = "DNS"

}

# calling the hosted zone

data "aws\_route53\_zone" "oyindamola" {

name = "oyindamola.gq"

private\_zone = false

}

# selecting validation method

resource "aws\_route53\_record" "oyindamola" {

for\_each = {

for dvo in aws\_acm\_certificate.oyindamola.domain\_validation\_options : dvo.domain\_name => {

name = dvo.resource\_record\_name

record = dvo.resource\_record\_value

type = dvo.resource\_record\_type

}

}

allow\_overwrite = true

name = each.value.name

records = [each.value.record]

ttl = 60

type = each.value.type

zone\_id = data.aws\_route53\_zone.oyindamola.zone\_id

}

# validate the certificate through DNS method

resource "aws\_acm\_certificate\_validation" "oyindamola" {

certificate\_arn = aws\_acm\_certificate.oyindamola.arn

validation\_record\_fqdns = [for record in aws\_route53\_record.oyindamola : record.fqdn]

}

# create records for tooling

resource "aws\_route53\_record" "tooling" {

zone\_id = data.aws\_route53\_zone.oyindamola.zone\_id

name = "tooling.oyindamola.gq"

type = "A"

alias {

name = aws\_lb.ext-alb.dns\_name

zone\_id = aws\_lb.ext-alb.zone\_id

evaluate\_target\_health = true

}

}

# create records for wordpress

resource "aws\_route53\_record" "wordpress" {

zone\_id = data.aws\_route53\_zone.oyindamola.zone\_id

name = "wordpress.oyindamola.gq"

type = "A"

alias {

name = aws\_lb.ext-alb.dns\_name

zone\_id = aws\_lb.ext-alb.zone\_id

evaluate\_target\_health = true

}

}

#### 3. Create an external (Internet facing) [Application Load Balancer (ALB)](https://docs.aws.amazon.com/elasticloadbalancing/latest/application/application-load-balancer-getting-started.html)

Create a file called alb.tf

First of all we will create the ALB, then we create the target group and lastly we will create the lsitener rule.

Useful Terraform Documentation, go through this documentation and understand the arguement needed for each resources:

* [ALB](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/lb)
* [ALB-target](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/lb_target_group)
* [ALB-listener](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/lb_listener)

We need to create an ALB to balance the traffic between the Instances:

resource "aws\_lb" "ext-alb" {

name = "ext-alb"

internal = false

security\_groups = [

aws\_security\_group.ext-alb-sg.id,

]

subnets = [

aws\_subnet.public[0].id,

aws\_subnet.public[1].id

]

tags = merge(

var.tags,

{

Name = "ACS-ext-alb"

},

)

ip\_address\_type = "ipv4"

load\_balancer\_type = "application"

}

To inform our ALB to where route the traffic we need to create a [Target Group](https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html) to point to its targets:

resource "aws\_lb\_target\_group" "nginx-tgt" {

health\_check {

interval = 10

path = "/healthstatus"

protocol = "HTTPS"

timeout = 5

healthy\_threshold = 5

unhealthy\_threshold = 2

}

name = "nginx-tgt"

port = 443

protocol = "HTTPS"

target\_type = "instance"

vpc\_id = aws\_vpc.main.id

}

Then we will need to create a [Listner](https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-listeners.html) for this target Group

resource "aws\_lb\_listener" "nginx-listner" {

load\_balancer\_arn = aws\_lb.ext-alb.arn

port = 443

protocol = "HTTPS"

certificate\_arn = aws\_acm\_certificate\_validation.oyindamola.certificate\_arn

default\_action {

type = "forward"

target\_group\_arn = aws\_lb\_target\_group.nginx-tgt.arn

}

}

Add the following outputs to output.tf to print them on screen

output "alb\_dns\_name" {

value = aws\_lb.ext-alb.dns\_name

}

output "alb\_target\_group\_arn" {

value = aws\_lb\_target\_group.nginx-tgt.arn

}

#### Create an Internal (Internal) [Application Load Balancer (ALB)](https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-internal-load-balancers.html)

For the Internal Load balancer we will fillow thje same concepts with the external load balancer.

Add the code snippets inside the alb.tf file

# ----------------------------

#Internal Load Balancers for webservers

#---------------------------------

resource "aws\_lb" "ialb" {

name = "ialb"

internal = true

security\_groups = [

aws\_security\_group.int-alb-sg.id,

]

subnets = [

aws\_subnet.private[0].id,

aws\_subnet.private[1].id

]

tags = merge(

var.tags,

{

Name = "ACS-int-alb"

},

)

ip\_address\_type = "ipv4"

load\_balancer\_type = "application"

}

To inform our ALB to where route the traffic we need to create a [Target Group](https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-target-groups.html) to point to its targets:

# --- target group for wordpress -------

resource "aws\_lb\_target\_group" "wordpress-tgt" {

health\_check {

interval = 10

path = "/healthstatus"

protocol = "HTTPS"

timeout = 5

healthy\_threshold = 5

unhealthy\_threshold = 2

}

name = "wordpress-tgt"

port = 443

protocol = "HTTPS"

target\_type = "instance"

vpc\_id = aws\_vpc.main.id

}

# --- target group for tooling -------

resource "aws\_lb\_target\_group" "tooling-tgt" {

health\_check {

interval = 10

path = "/healthstatus"

protocol = "HTTPS"

timeout = 5

healthy\_threshold = 5

unhealthy\_threshold = 2

}

name = "tooling-tgt"

port = 443

protocol = "HTTPS"

target\_type = "instance"

vpc\_id = aws\_vpc.main.id

}

Then we will need to create a [Listner](https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-listeners.html) for this target Group

# For this aspect a single listener was created for the wordpress which is default,

# A rule was created to route traffic to tooling when the host header changes

resource "aws\_lb\_listener" "web-listener" {

load\_balancer\_arn = aws\_lb.ialb.arn

port = 443

protocol = "HTTPS"

certificate\_arn = aws\_acm\_certificate\_validation.oyindamola.certificate\_arn

default\_action {

type = "forward"

target\_group\_arn = aws\_lb\_target\_group.wordpress-tgt.arn

}

}

# listener rule for tooling target

resource "aws\_lb\_listener\_rule" "tooling-listener" {

listener\_arn = aws\_lb\_listener.web-listener.arn

priority = 99

action {

type = "forward"

target\_group\_arn = aws\_lb\_target\_group.tooling-tgt.arn

}

condition {

host\_header {

values = ["tooling.oyindamola.gq"]

}

}

}

# CREATING AUSTOALING GROUPS

#### This Section we will create the [Auto Scaling Group (ASG)](https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html)

Now we need to configure our ASG to be able to scale the EC2s out and in depending on the application traffic.

Before we start configuring an ASG, we need to create the launch template and the the AMI needed. For now we are going to use a random AMI from AWS, then in project 19, we will use [Packer](https://www.packer.io/intro)to create our ami.

Based on our Architetcture we need for Auto Scaling Groups for bastion, nginx, wordpress and tooling, so we will create two files; asg-bastion-nginx.tf will contain Launch Template and Austoscaling froup for Bastion and Nginx, then asg-wordpress-tooling.tf will contain Launch Template and Austoscaling group for wordpress and tooling.

Useful Terraform Documentation, go through this documentation and understand the arguement needed for each resources:

* [SNS-topic](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/sns_topic)
* [SNS-notification](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/autoscaling_notification)
* [Austoscaling](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/autoscaling_group)
* [Launch-template](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/launch_template)

Create asg-bastion-nginx.tf and paste all the code snippet below;

#### creating sns topic for all the auto scaling groups

resource "aws\_sns\_topic" "david-sns" {

name = "Default\_CloudWatch\_Alarms\_Topic"

}

#### creating notification for all the auto scaling groups

resource "aws\_autoscaling\_notification" "david\_notifications" {

group\_names = [

aws\_autoscaling\_group.bastion-asg.name,

aws\_autoscaling\_group.nginx-asg.name,

aws\_autoscaling\_group.wordpress-asg.name,

aws\_autoscaling\_group.tooling-asg.name,

]

notifications = [

"autoscaling:EC2\_INSTANCE\_LAUNCH",

"autoscaling:EC2\_INSTANCE\_TERMINATE",

"autoscaling:EC2\_INSTANCE\_LAUNCH\_ERROR",

"autoscaling:EC2\_INSTANCE\_TERMINATE\_ERROR",

]

topic\_arn = aws\_sns\_topic.david-sns.arn

}

# launch template for bastion

resource "random\_shuffle" "az\_list" {

input = data.aws\_availability\_zones.available.names

}

resource "aws\_launch\_template" "bastion-launch-template" {

image\_id = var.ami

instance\_type = "t2.micro"

vpc\_security\_group\_ids = [aws\_security\_group.bastion\_sg.id]

iam\_instance\_profile {

name = aws\_iam\_instance\_profile.ip.id

}

key\_name = var.keypair

placement {

availability\_zone = "random\_shuffle.az\_list.result"

}

lifecycle {

create\_before\_destroy = true

}

tag\_specifications {

resource\_type = "instance"

tags = merge(

var.tags,

{

Name = "bastion-launch-template"

},

)

}

user\_data = filebase64("${path.module}/bastion.sh")

}

# ---- Autoscaling for bastion hosts

resource "aws\_autoscaling\_group" "bastion-asg" {

name = "bastion-asg"

max\_size = 2

min\_size = 1

health\_check\_grace\_period = 300

health\_check\_type = "ELB"

desired\_capacity = 1

vpc\_zone\_identifier = [

aws\_subnet.public[0].id,

aws\_subnet.public[1].id

]

launch\_template {

id = aws\_launch\_template.bastion-launch-template.id

version = "$Latest"

}

tag {

key = "Name"

value = "bastion-launch-template"

propagate\_at\_launch = true

}

}

# launch template for nginx

resource "aws\_launch\_template" "nginx-launch-template" {

image\_id = var.ami

instance\_type = "t2.micro"

vpc\_security\_group\_ids = [aws\_security\_group.nginx-sg.id]

iam\_instance\_profile {

name = aws\_iam\_instance\_profile.ip.id

}

key\_name = var.keypair

placement {

availability\_zone = "random\_shuffle.az\_list.result"

}

lifecycle {

create\_before\_destroy = true

}

tag\_specifications {

resource\_type = "instance"

tags = merge(

var.tags,

{

Name = "nginx-launch-template"

},

)

}

user\_data = filebase64("${path.module}/nginx.sh")

}

# ------ Autoscslaling group for reverse proxy nginx ---------

resource "aws\_autoscaling\_group" "nginx-asg" {

name = "nginx-asg"

max\_size = 2

min\_size = 1

health\_check\_grace\_period = 300

health\_check\_type = "ELB"

desired\_capacity = 1

vpc\_zone\_identifier = [

aws\_subnet.public[0].id,

aws\_subnet.public[1].id

]

launch\_template {

id = aws\_launch\_template.nginx-launch-template.id

version = "$Latest"

}

tag {

key = "Name"

value = "nginx-launch-template"

propagate\_at\_launch = true

}

}

# attaching autoscaling group of nginx to external load balancer

resource "aws\_autoscaling\_attachment" "asg\_attachment\_nginx" {

autoscaling\_group\_name = aws\_autoscaling\_group.nginx-asg.id

alb\_target\_group\_arn = aws\_lb\_target\_group.nginx-tgt.arn

}

Autoscaling for wordpres and toolibng will be created in a seperate file

Create asg-wordpress-tooling.tf and paste the following code

# launch template for wordpress

resource "aws\_launch\_template" "wordpress-launch-template" {

image\_id = var.ami

instance\_type = "t2.micro"

vpc\_security\_group\_ids = [aws\_security\_group.webserver-sg.id]

iam\_instance\_profile {

name = aws\_iam\_instance\_profile.ip.id

}

key\_name = var.keypair

placement {

availability\_zone = "random\_shuffle.az\_list.result"

}

lifecycle {

create\_before\_destroy = true

}

tag\_specifications {

resource\_type = "instance"

tags = merge(

var.tags,

{

Name = "wordpress-launch-template"

},

)

}

user\_data = filebase64("${path.module}/wordpress.sh")

}

# ---- Autoscaling for wordpress application

resource "aws\_autoscaling\_group" "wordpress-asg" {

name = "wordpress-asg"

max\_size = 2

min\_size = 1

health\_check\_grace\_period = 300

health\_check\_type = "ELB"

desired\_capacity = 1

vpc\_zone\_identifier = [

aws\_subnet.private[0].id,

aws\_subnet.private[1].id

]

launch\_template {

id = aws\_launch\_template.wordpress-launch-template.id

version = "$Latest"

}

tag {

key = "Name"

value = "wordpress-asg"

propagate\_at\_launch = true

}

}

# attaching autoscaling group of wordpress application to internal loadbalancer

resource "aws\_autoscaling\_attachment" "asg\_attachment\_wordpress" {

autoscaling\_group\_name = aws\_autoscaling\_group.wordpress-asg.id

alb\_target\_group\_arn = aws\_lb\_target\_group.wordpress-tgt.arn

}

# launch template for toooling

resource "aws\_launch\_template" "tooling-launch-template" {

image\_id = var.ami

instance\_type = "t2.micro"

vpc\_security\_group\_ids = [aws\_security\_group.webserver-sg.id]

iam\_instance\_profile {

name = aws\_iam\_instance\_profile.ip.id

}

key\_name = var.keypair

placement {

availability\_zone = "random\_shuffle.az\_list.result"

}

lifecycle {

create\_before\_destroy = true

}

tag\_specifications {

resource\_type = "instance"

tags = merge(

var.tags,

{

Name = "tooling-launch-template"

},

)

}

user\_data = filebase64("${path.module}/tooling.sh")

}

# ---- Autoscaling for tooling -----

resource "aws\_autoscaling\_group" "tooling-asg" {

name = "tooling-asg"

max\_size = 2

min\_size = 1

health\_check\_grace\_period = 300

health\_check\_type = "ELB"

desired\_capacity = 1

vpc\_zone\_identifier = [

aws\_subnet.private[0].id,

aws\_subnet.private[1].id

]

launch\_template {

id = aws\_launch\_template.tooling-launch-template.id

version = "$Latest"

}

tag {

key = "Name"

value = "tooling-launch-template"

propagate\_at\_launch = true

}

}

# attaching autoscaling group of tooling application to internal loadbalancer

resource "aws\_autoscaling\_attachment" "asg\_attachment\_tooling" {

autoscaling\_group\_name = aws\_autoscaling\_group.tooling-asg.id

alb\_target\_group\_arn = aws\_lb\_target\_group.tooling-tgt.arn

}

# STORAGE AND DATABASE

Useful Terraform Documentation, go through this documentation and understand the arguement needed for each resources:

* [RDS](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/db_subnet_group)
* [EFS](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/efs_file_system)
* [KMS](https://registry.terraform.io/providers/hashicorp/aws/latest/docs/resources/kms_key)

#### Create Elastic File System (EFS)

In order to create an EFS you need to create a [KMS key](https://aws.amazon.com/kms/getting-started/).

AWS Key Management Service (KMS) makes it easy for you to create and manage cryptographic keys and control their use across a wide range of AWS services and in your applications.

Add the following code to efs.tf

# create key from key management system

resource "aws\_kms\_key" "ACS-kms" {

description = "KMS key "

policy = <<EOF

{

"Version": "2012-10-17",

"Id": "kms-key-policy",

"Statement": [

{

"Sid": "Enable IAM User Permissions",

"Effect": "Allow",

"Principal": { "AWS": "arn:aws:iam::${var.account\_no}:user/segun" },

"Action": "kms:\*",

"Resource": "\*"

}

]

}

EOF

}

# create key alias

resource "aws\_kms\_alias" "alias" {

name = "alias/kms"

target\_key\_id = aws\_kms\_key.ACS-kms.key\_id

}

Let us create EFS and it mount targets- add the following code to efs.tf

# create Elastic file system

resource "aws\_efs\_file\_system" "ACS-efs" {

encrypted = true

kms\_key\_id = aws\_kms\_key.ACS-kms.arn

tags = merge(

var.tags,

{

Name = "ACS-efs"

},

)

}

# set first mount target for the EFS

resource "aws\_efs\_mount\_target" "subnet-1" {

file\_system\_id = aws\_efs\_file\_system.ACS-efs.id

subnet\_id = aws\_subnet.private[2].id

security\_groups = [aws\_security\_group.datalayer-sg.id]

}

# set second mount target for the EFS

resource "aws\_efs\_mount\_target" "subnet-2" {

file\_system\_id = aws\_efs\_file\_system.ACS-efs.id

subnet\_id = aws\_subnet.private[3].id

security\_groups = [aws\_security\_group.datalayer-sg.id]

}

# create access point for wordpress

resource "aws\_efs\_access\_point" "wordpress" {

file\_system\_id = aws\_efs\_file\_system.ACS-efs.id

posix\_user {

gid = 0

uid = 0

}

root\_directory {

path = "/wordpress"

creation\_info {

owner\_gid = 0

owner\_uid = 0

permissions = 0755

}

}

}

# create access point for tooling

resource "aws\_efs\_access\_point" "tooling" {

file\_system\_id = aws\_efs\_file\_system.ACS-efs.id

posix\_user {

gid = 0

uid = 0

}

root\_directory {

path = "/tooling"

creation\_info {

owner\_gid = 0

owner\_uid = 0

permissions = 0755

}

}

}

#### Create [MySQL RDS](https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_MySQL.html)

Let us create the RDS itself using this snippet of code in rds.tf file:

# This section will create the subnet group for the RDS instance using the private subnet

resource "aws\_db\_subnet\_group" "ACS-rds" {

name = "acs-rds"

subnet\_ids = [aws\_subnet.private[2].id, aws\_subnet.private[3].id]

tags = merge(

var.tags,

{

Name = "ACS-rds"

},

)

}

# create the RDS instance with the subnets group

resource "aws\_db\_instance" "ACS-rds" {

allocated\_storage = 20

storage\_type = "gp2"

engine = "mysql"

engine\_version = "5.7"

instance\_class = "db.t2.micro"

name = "daviddb"

username = var.master-username

password = var.master-password

parameter\_group\_name = "default.mysql5.7"

db\_subnet\_group\_name = aws\_db\_subnet\_group.ACS-rds.name

skip\_final\_snapshot = true

vpc\_security\_group\_ids = [aws\_security\_group.datalayer-sg.id]

multi\_az = "true"

}

Before Applying, if you take note, we gave refrence to a lot of varibales in our resources that has not been declared in the variables.tf file. Go through the entire code and spot this variables and declare them in the variables.tf file.

If you have done that well, you file should like this one below.

variable "region" {

type = string

description = "The region to deploy resources"

}

variable "vpc\_cidr" {

type = string

description = "The VPC cidr"

}

variable "enable\_dns\_support" {

type = bool

}

variable "enable\_dns\_hostnames" {

dtype = bool

}

variable "enable\_classiclink" {

type = bool

}

variable "enable\_classiclink\_dns\_support" {

type = bool

}

variable "preferred\_number\_of\_public\_subnets" {

type = number

description = "Number of public subnets"

}

variable "preferred\_number\_of\_private\_subnets" {

type = number

description = "Number of private subnets"

}

variable "name" {

type = string

default = "ACS"

}

variable "tags" {

description = "A mapping of tags to assign to all resources."

type = map(string)

default = {}

}

variable "ami" {

type = string

description = "AMI ID for the launch template"

}

variable "keypair" {

type = string

description = "key pair for the instances"

}

variable "account\_no" {

type = number

description = "the account number"

}

variable "master-username" {

type = string

description = "RDS admin username"

}

variable "master-password" {

type = string

description = "RDS master password"

}

Now, we are almost done but we need to update the last file which is terraform.tfvars file. In this file we are going to declare the values for the variables in our varibales.tf file.

Open the terraform.tfvars file and add the code below

region = "us-east-1"

vpc\_cidr = "172.16.0.0/16"

enable\_dns\_support = "true"

enable\_dns\_hostnames = "true"

enable\_classiclink = "false"

enable\_classiclink\_dns\_support = "false"

preferred\_number\_of\_public\_subnets = "2"

preferred\_number\_of\_private\_subnets = "4"

environment = "production"

ami = "ami-0b0af3577fe5e3532"

keypair = "devops"

# Ensure to change this to your acccount number

account\_no = "123456789"

db-username = "david"

db-password = "devopspbl"

tags = {

Enviroment = "production"

Owner-Email = "infradev-segun@darey.io"

Managed-By = "Terraform"

Billing-Account = "1234567890"

}

At this point, you shall have pretty much all infrastructure elements ready to be deployed automatically, but before we paln and apply our code we need to take note of two things;

* we have a long list of files which may looks confusing but that is not bad for a start, we are going to fix this using the concepts of modules in Project 18
* Secondly, our application wont work becuase in out shell script that was passed into the launch some endpoints like the RDs and EFS point is needed in which they have not been created yet. So in project 19 we will use our Ansible knowledge to fix this.

Try to plan and apply your Terraform codes, explore the resources in AWS console and make sure you destroy them right away to avoid massive costs.

### Additional tasks

In addition to regular project submission include following:

1. Summarise your understanding on Networking concepts like **IP Address, Subnets, CIDR Notation, IP Routing, Internet Gateways, NAT**
2. Summarise your understanding of the [OSI Model](https://en.wikipedia.org/wiki/OSI_model), [TCP/IP suite](https://en.wikipedia.org/wiki/Internet_protocol_suite) and how [they are connected](https://en.wikipedia.org/wiki/Internet_protocol_suite#Comparison_of_TCP/IP_and_OSI_layering) – research beyond the provided articles, watch different YouTube videos to fully understand the concept around OSI and how it is related to the Internet and end-to-end Web Solutions. You don not need to memorise the layers – just understand the idea around it.
3. Explain the difference between assume role policy and role policy

### Congratulations!

Now you have fully automated creation of AWS Infrastructure for 2 websites with Terraform. In the next project we will further enhance our codes by refactoring and introducing more exciting Terraform concepts! Go ahead and continue your PBL journey with us!

