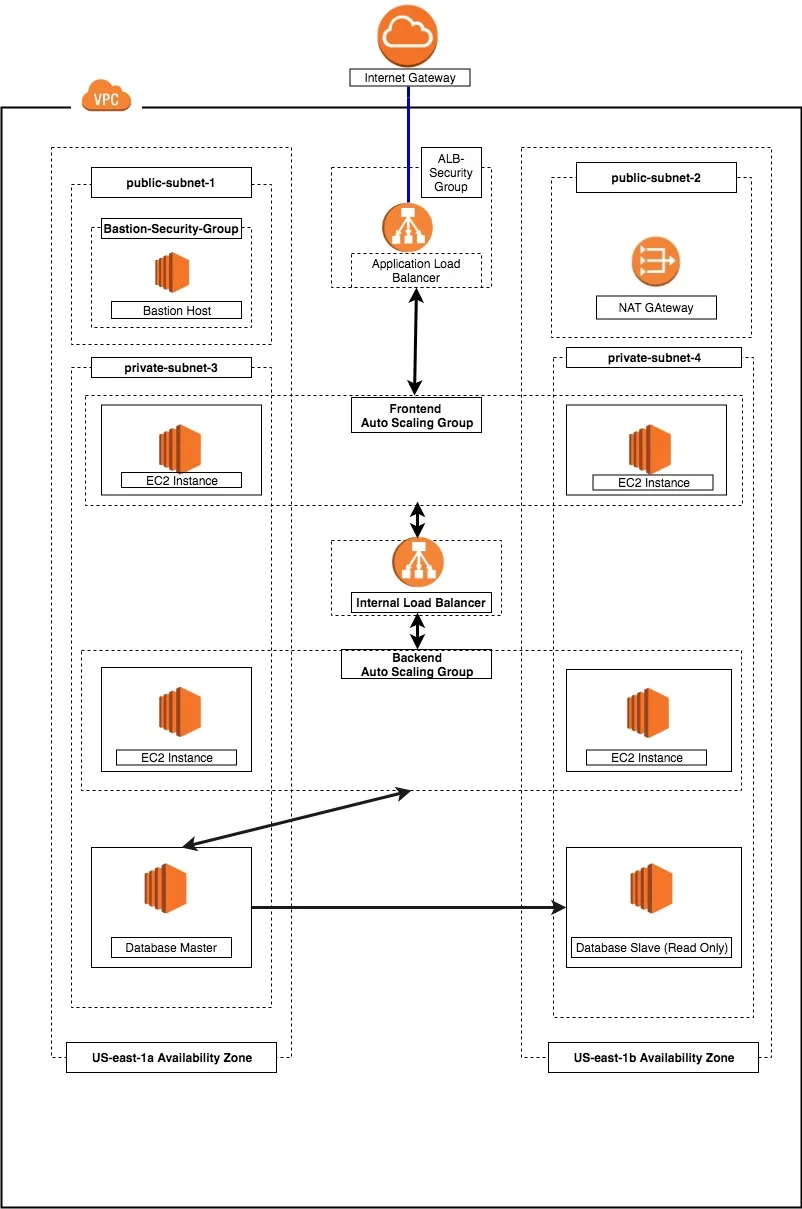
**Three-Tier Architecture in AWS**

Introduction:

A three-tier architecture is a software architecture pattern where the application is broken down into three logical tiers: the presentation layer, the business logic layer and the data storage layer. This architecture is used in a client-server application such as a web application that has the frontend, the backend and the database. Each of these layers or tiers does a specific task and can be managed independently of each other. This a shift from the monolithic way of building an application where the frontend, the backend and the database are both sitting in one place.

Amazon Web Service (AWS) is a cloud platform that provides different cloud computing services to their customers. To view a list of all AWS services and products, click on this [link](https://aws.amazon.com/" \t "_blank). In this article, we shall be making use of the following AWS services to design and build a three-tier cloud infrastructure: [Elastic Compute Cloud (EC2)](https://aws.amazon.com/ec2/?hp=tile&so-exp=below" \t "_blank), [Auto Scaling Group](https://aws.amazon.com/ec2/autoscaling/?hp=tile&so-exp=below" \t "_blank), [Virtual Private Cloud](https://aws.amazon.com/vpc/?hp=tile&so-exp=below" \t "_blank)(VPC), [Elastic Load Balancer (ELB)](https://aws.amazon.com/elasticloadbalancing/?hp=tile&so-exp=below" \t "_blank), [Security Groups](https://docs.aws.amazon.com/vpc/latest/userguide/VPC_SecurityGroups.html" \t "_blank) and the [Internet Gateway](https://docs.aws.amazon.com/vpc/latest/userguide/VPC_Internet_Gateway.html" \t "_blank). Our infrastructure will be designed to be highly available and fault tolerant.



**solving process:**

1. **Modularity**: The essence of having a three-tier architecture is to modularize our application such that each part can be managed independently of each other. With modularity, teams can focus on different tiers of the application and changes made as quickly as possible. Also, modularization helps us recover quickly from an unexpected disaster by focusing solely on the faulty part.
2. **Scalability**: Each tier of the architecture can scale horizontally to support the traffic and request demand coming to it. This can easily be done by adding more EC2 instances to each tier and load balancing across them. For instance, assuming we have two EC2 instances serving our backend application and each of the EC2 instances is working at 80% CPU utilization, we can easily scale the backend tier by adding more EC2 instances to it so that the load can be distributed. We can also automatically reduce the number of the EC2 instances when the load is less.
3. **High Availability**: With the traditional data centre, our application is sitting in one geographical location. If there is an earthquake, flooding or even power outage in that location where our application is hosted, our application will not be available. With AWS, we can design our infrastructure to be highly available by hosting our application in different locations known as the [availability zones](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html" \t "_blank).
4. **Fault Tolerant**: We want our infrastructure to comfortably adapt to any unexpected change both to traffic and fault. This is usually done by adding a redundant system that will account for such a hike in traffic when it does occur. So instead of having two EC2 instances working at 50% each, such that when one instance goes bad, the other instance will be working at 100% capacity until a new instance is brought up by our Auto Scaling Group, we have extra instance making it three instances working at approximately 35% each. This is usually a tradeoff made against the cost of setting up a redundant system.
5. **Security**: We want to design an infrastructure that is highly secured and protected from the prying eyes of hackers. As much as possible, we want to avoid exposing our interactions within the application over the internet. This simply means that the application will communicate within themselves with a private IP. The presentation (frontend) tier of the infrastructure will be in a private subnet (the subnet with no public IP assigned to its instances) within the VPC. Users can only reach the frontend through the application load balancer. The backend and the database tier will also be in the private subnet because we do not want to expose them over the internet. We will set up the Bastion host for remote SSH and a NAT gateway for our private subnets to access the internet. The AWS security group helps us limit access to our infrastructure setup.

**Before we get started**

To follow along, you need to have an [AWS](http://console.aws.amazon.com/" \t "_blank) account. We shall be making use of the AWS free-tier resources so we do not incur charges while learning.

**Note**: At the end of this tutorial, you need to stop and delete all the resources such as the EC2 instances, Auto Scaling Group, Elastic Load Balancer etc you set up. Otherwise, you get charged for it when you keep them running for a long.

**Let’s Begin**

1. **Setup the Virtual Private Cloud (VPC):**VPC stands for Virtual Private Cloud (VPC). It is a virtual network where you create and manage your AWS resource in a more secure and scalable manner. Go to the VPC section of the AWS services, and click on the **Create VPC** button.

Give your VPC a name and a [CIDR](https://searchnetworking.techtarget.com/definition/CIDR" \t "_blank) block of 10.0.0.0/16

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

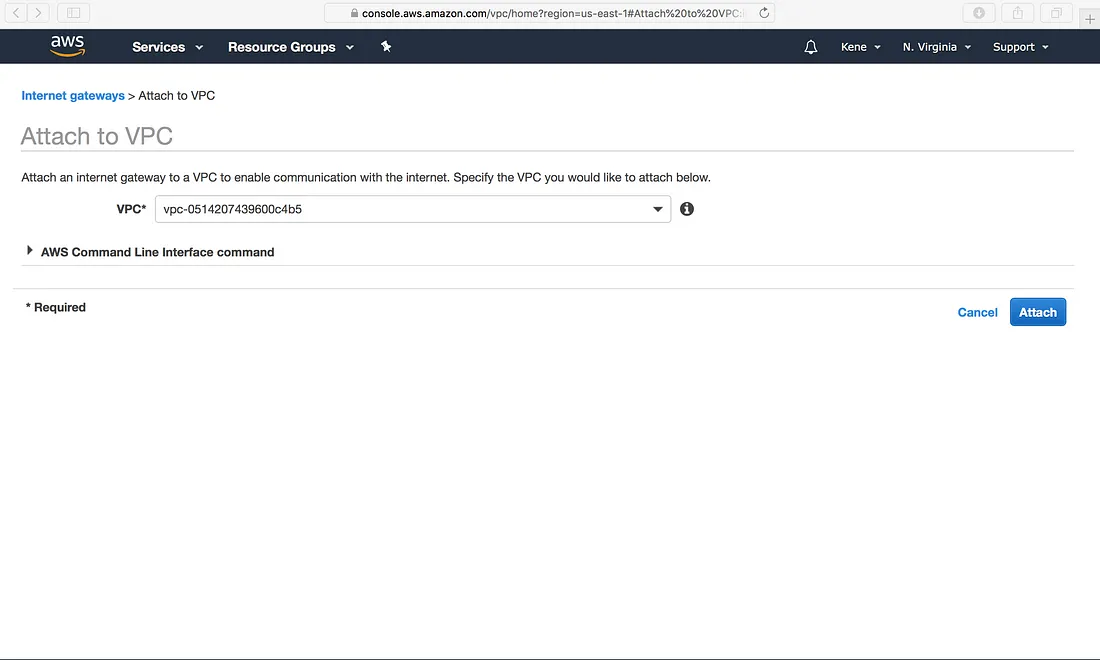
**Create Internet Gateway**

We need to attach our VPC to the internet gateway. To do that:

**a.** we select the internet gateway

**b.** Click on the **Actions** button and then select Attach to VPC.

**c.** Select the VPC to attach the internet gateway and click **Attach**

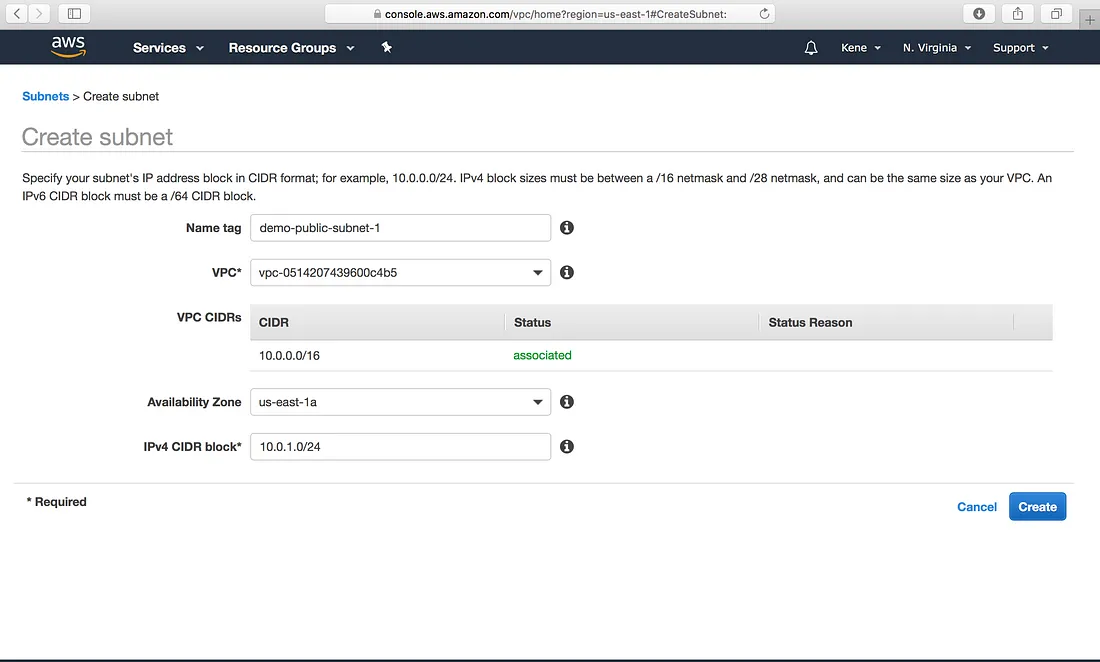


**Attach the VPC to the internet gateway**

**3. Create 4 Subnets:**The subnet is a way for us to group our resources within the VPC with their IP range. A subnet can be public or private. EC2 instances within a public subnet have public IPs and can directly access the internet while those in the private subnet does not have public IPs and can only access the internet through a [NAT](https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html" \t "_blank) gateway.

For our setup, we shall be creating the following subnets with the corresponding IP ranges.

* demo-public-subnet-1 | CIDR (10.0.1.0/24) | Availability Zone (us-east-1a)
* demo-public-subnet-2 | CIDR (10.0.2.0/24) | Availability Zone (us-east-1b)
* demo-private-subnet-3 | CIDR (10.0.3.0/24) | Availability Zone (us-east-1a)
* demo-private-subnet-4 | CIDR(10.0.4.0/24) | Availability Zone (us-east-1b)



A screenshot of a computer

Description automatically generated

**four subnets in our VPC**

**4. Create Two Route Tables:**Route tables is a set of rule that determines how data moves within our network. We need two route tables; private route table and public route table. The public route table will define which subnets that will have direct access to the internet ( ie public subnets) while the private route table will define which subnet goes through the NAT gateway (ie private subnet).

To create route tables, navigate over to the **Route Tables** page and click on **Create route table** button.

A screenshot of a computer

Description automatically generated

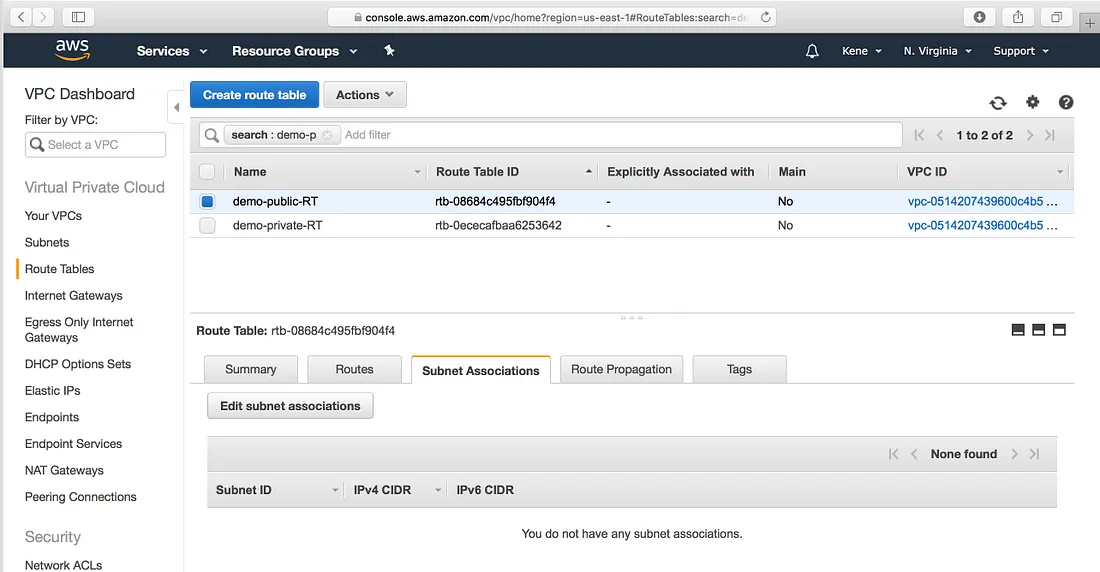
A screenshot of a computer

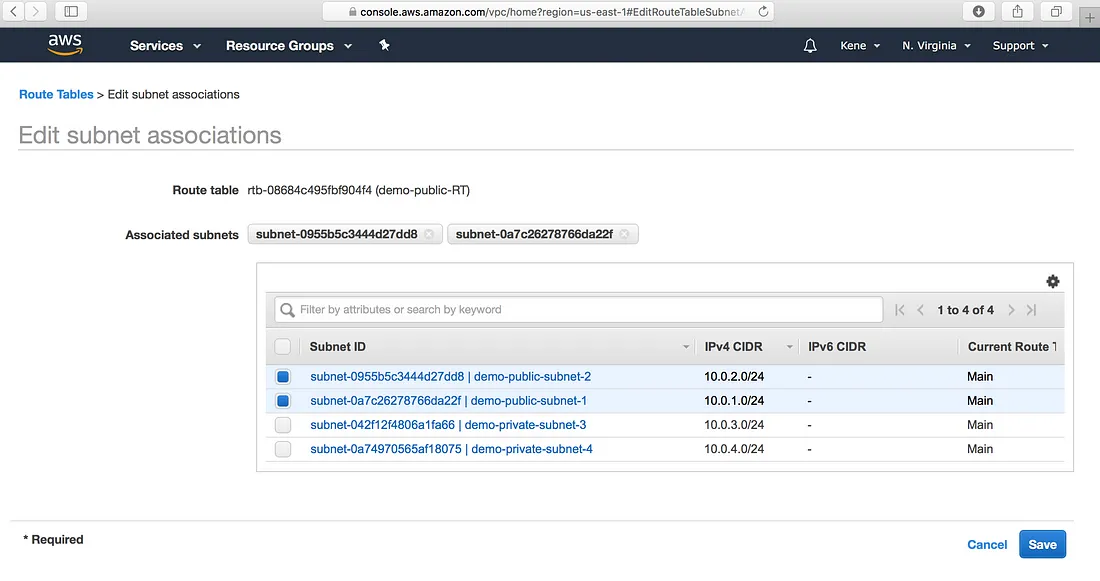
Description automatically generated

**Private and Public Route Tables**

The public and the private subnet needs to be associated with the public and the private route table respectively.

To do that, we select the route table and then choose the **Subnet Association** tab.





**Select the public subnet for the public route table**

We also need to route the traffic to the internet through the **internet gateway**for our public route table.

To do that we select the public route table and then choose the **Routes** tab. The rule should be similar to the one shown below:



Edit Route for the public route table

**5. Create the NAT Gateway:**The NAT gateway enables the EC2 instances in the private subnet to access the internet. The NAT Gateway is an AWS managed service for the NAT instance. To create the NAT gateway, navigate to the NAT Gateways page, and then click on the **Create NAT Gateway.**

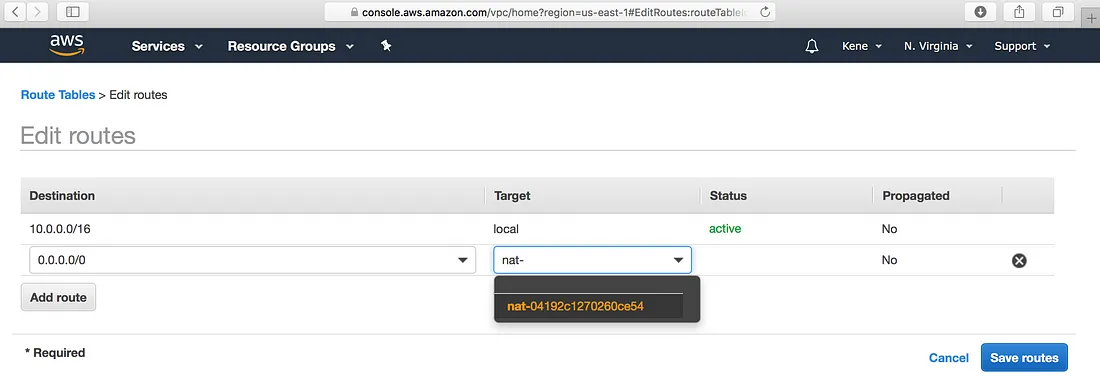
Please ensure that you know the Subnet ID for the **demo-public-subnet-2.**This will be needed when creating the NAT gateway.

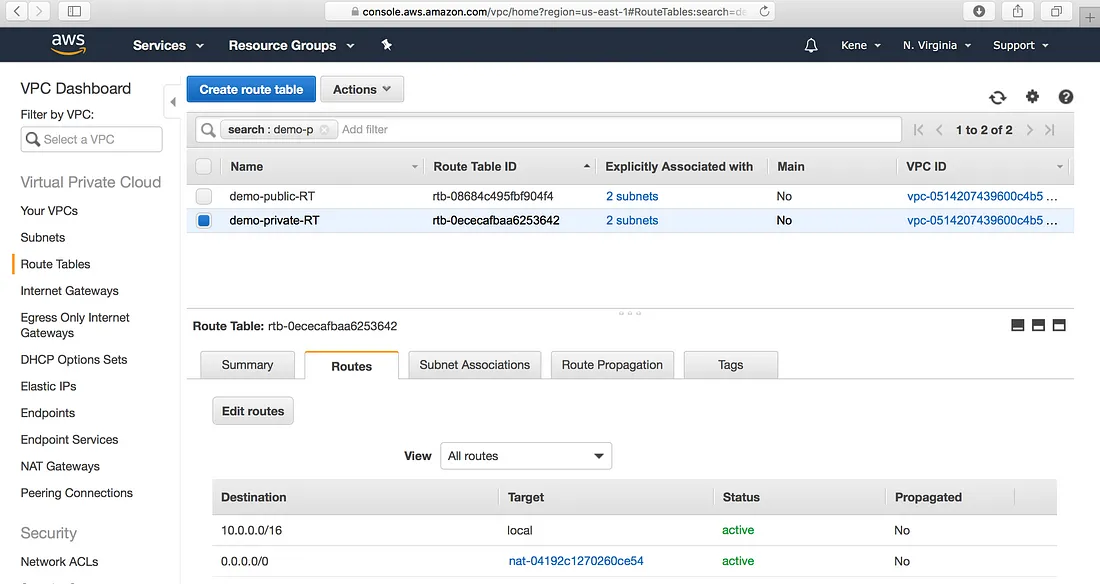
A screenshot of a computer

Description automatically generated

**Create NAT Gateway**

Now that we have the NAT gateway, we are going to edit the private route table to make use of the NAT gateway to access the internet.





Edit Private Route Table to use NAT Gateway for private EC2 instances

**6**. **Create Elastic Load Balancer:** From our architecture, our frontend tier can only accept traffic from the elastic load balancer which connects directly with the internet gateway while our backend tier will receive traffic through the internal load balancer. The essence of the load balancer is to distribute load across the EC2 instances serving that application. If however, the application is using sessions, then the application needs to be rewritten such that sessions can be stored in either the Elastic Cache or the DynamoDB. To create the two load balancers needed in our architecture, we navigate to the **Load Balancer** page and click on **Create Load Balancer.**

**a.** Select the Application Load Balancer.

A screenshot of a computer

Description automatically generated

Select Application Load Balancer

**b.**Click on the **Create** button

**c.** Configure the Load Balancer with a name. Select **internet facing** for the load balancer that we will use to communicate with the frontend and **internal**for the one we will use for our backend.

**Internet Facing Load Balancer for the Frontend tier**

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Internal Load Balancer for the Backend Tier**

**d.** Under the Availability Zone, for the **internet facing Load Balancer**, we will select the two **public subnets** while for our**internal Load Balancer**, we will select the two **private subnet.**

A screenshot of a computer

Description automatically generatedAvailability Zone for the internal Load Balancer

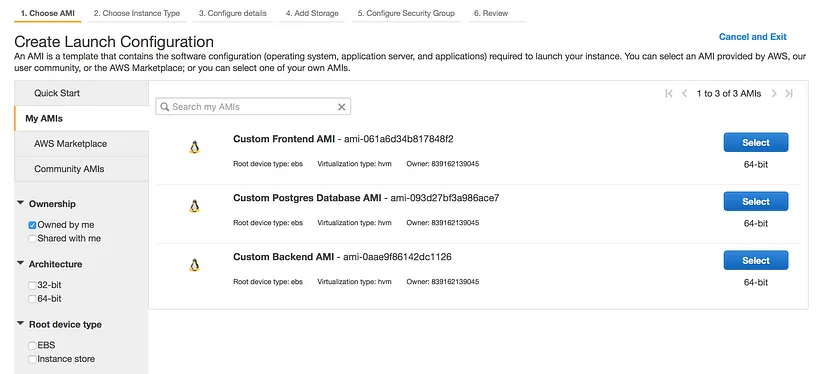
**e.**Under the Security Group, we **only need**to allow ports that the application needs. For instance, we need to allow **HTTP port 80 and/or HTTPS port 443** on our **internet facing load balancer**. For the **internal load balancer**, we only open the port that the backend runs on (eg: port 3000) and the make such port **only open to the security group of the frontend**. This will allow only the frontend to have access to that port within our architecture.

**f.** Under the **Configure Routing,**we need to configure our Target Group to have the **Target type** of **instance.**We will give the **Target Group** a name that will enable us to identify it. This is will be needed when we will create our **Auto Scaling Group.**For example, we can name the Target Group of our frontend to be **Demo-Frontend-TG**

Skip the Register Targets and then go ahead and review the configuration and then click on the **Create** button.

**7. Auto Scaling Group:**We can simply create like two EC2 instances and directly attach these EC2 instances to our load balancer. The problem with that is that our application will no longer scale to accommodate traffic or shrink when there is no traffic to save cost. With Auto Scaling Group, we can achieve this feat. Auto Scaling Group is can automatically adjust the size of the EC2 instances serving the application based on need. This is what makes it a good approach rather than directly attaching the EC2 instances to the load balancer.

To create an Auto Scaling Group, navigate to the **Auto Scaling Group** page, Click on the **Create Auto Scaling Group** button.

1. Auto Scaling Group needs to have a common configuration that instances within it **MUST**have. This common configuration is made possible with the help of the **Launch Configuration**. In our Launch configuration, under the Choose AMI, the best practice is to choose the AMI which contains the application and its dependencies bundled together. You can also create your Custom AMI for each tier of our application

**b.** Choose the appropriate instance type. For a demo, I recommend you choose t2.micro (free tier eligible) so that you do not incur charges.

**c.** Under the Configure details, give the Launch Configuration a name, eg **Demo-Frontend-LC.**Also, under the **Advance Details**dropdown, the **User data**is provided for you to type in a command that is needed to install dependencies and start the application.

A screenshot of a computer

Description automatically generated

**d.** Again under the security group, we want to only allow the ports that are necessary for our application.

**e.** Review the Configuration and Click on **Create Launch Configuration** button. Go ahead and create a new key pair. Ensure you download it before proceeding.

**f.** Now we have our Launch Configuration, we can finish up with the creating our Auto Scaling Group. Use the below image as a template for setting up yours.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Auto Scaling Group 1

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

Auto Scaling Group 2

**g.** Under the Configure scaling policies, we want to add one instance when the CPU is greater than or equal to 80% and to scale down when the CPU is less than or equal to 50%. Use the image as a template.

Scale-up

A screenshot of a computer

Description automatically generated

Scale Down

A screenshot of a computer

Description automatically generated**h.** We can now go straight to Review and then Click on the **Create Auto Scaling group** button. This process is to be done for both the frontend tier and the backend tier but not the data storage tier.

We have almost setup or architecture. However, we cannot SSH into the EC2 instances in the private subnet. This is because have not created our bastion host. So the last part of this article will show how to create the bastion host.

**8. Bastion Host:**The bastion host is just an EC2 instance that sits in the public subnet. The best practice is to only allow SSH to this instance from your trusted IP. To create a bastion host, navigate to the EC2 instance page and create an EC2 instance in the **demo-public-subnet-1**subnet within our VPC. Also, ensure that it has public IP.

Security Group of the Bastion Host

We also need to allow SSH from our private instances from the Bastion Host.

A screenshot of a computer

Description automatically generated

**Conclusion**

There were lots of clicking and configurations when using the console to set up a three-tier architecture in AWS. It is, however, necessary that a beginner goes through this procedure before moving towards automation.

In our next article, we will automate this whole architecture using terraform

**Thankyou!**