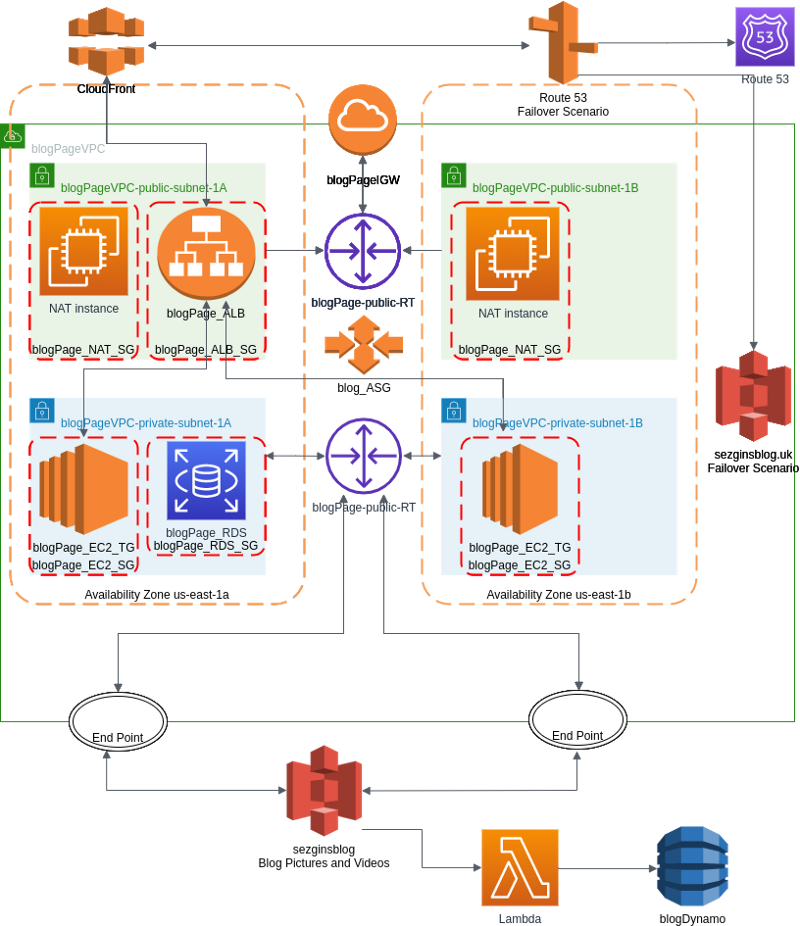
### **Django Blog Page App deployed on AWS ALB with Auto Scaling, S3, RDS, VPC’s Components, DynamoDB and CloudFront with Route 53**



### Description

Blog Page Application aims to deploy blog application as a web application written Django Framework on AWS Cloud Infrastructure. This infrastructure has an Application Load Balancer with Auto Scaling Group of Elastic Compute Cloud (EC2) Instances and Relational Database Service (RDS) on defined VPC. Also, The CloudFront and Route 53 services are located in front of the architecture and manage the traffic insecure. The user is able to upload pictures and videos on their own blog page and these are kept on S3 Bucket. This architecture will be created by Firms DevOps Guy.

### Problem Statement

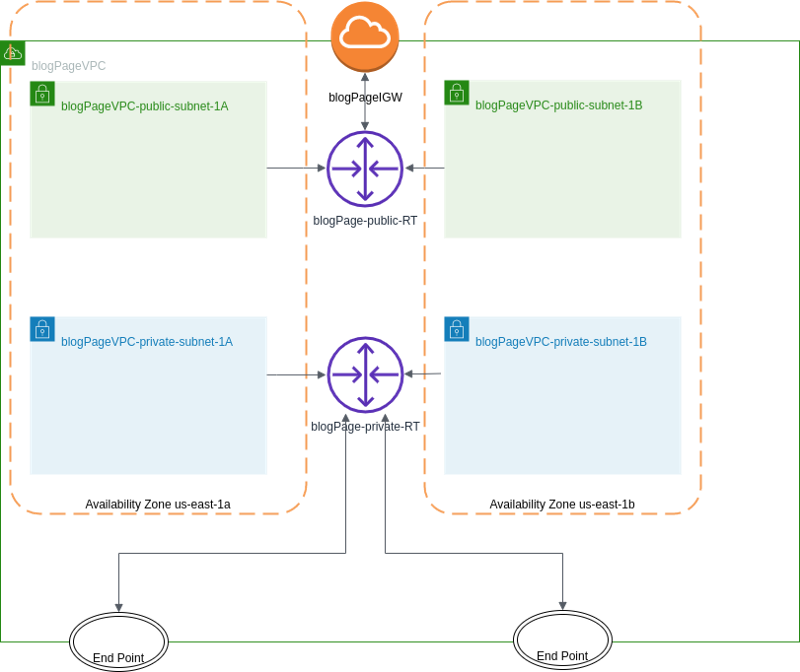
Your company has recently ended up a project that aims to serve as a Blog web application on an isolated VPC environment. You and your colleagues have started to work on the project. Your Developer team has developed the application and you are going to deploy the app in the production environment.

Application is coded by the Fullstack development team and given to you as the DevOps team. The app allows users to write their own blog page to whom user registration data should be kept in a separate MySQL database in AWS RDS service and pictures or videos should be kept in the S3 bucket. The object list of S3 Bucket containing movies and videos is recorded on the DynamoDB table.

The web application will be deployed using the Django framework. The web application should be accessible via web browser from anywhere insecure. You are requested to push your program to the project repository on GitHub. You are going to pull it into the web servers in the production environment on AWS Cloud.

### Steps to Solution

#### Step 1: Create dedicated VPC and whole components



First of all, I created a custom VPC named `blogPageVPC`. This VPC’s CIDR blok is `90.90.0.0/16`. I enabled the `DNS hostnames` for the `blogPageVPC`. I created my VPC in two AZs. I created a public subnet named `blogPageVPC-public-subnet-1A` under the `blogPageVPC` in AZ us-east-1a with 90.90.10.0/24. I created a private subnet named `blogPageVPC-private-subnet-1A` under the `blogPageVPC` in AZ us-east-1a with 90.90.11.0/24. I created a public subnet named `blogPageVPC-public-subnet-1B` under the `blogPageVPC` in AZ us-east-1b with 90.90.20.0/24. I created a private subnet named `blogPageVPC-private-subnet-1B` under the `blogPageVPC` in AZ us-east-1b with 90.90.21.0/24. I set set `auto-assign IP` up for public subnets.

AWS assigns public IPs. With this isolated network solution, I placed different resources in subnets and secured them, I gathered a common resource cluster under different subnets. In addition, in failover scenarios, you can create replicas for your other instances by distributing your resources to different subnets.

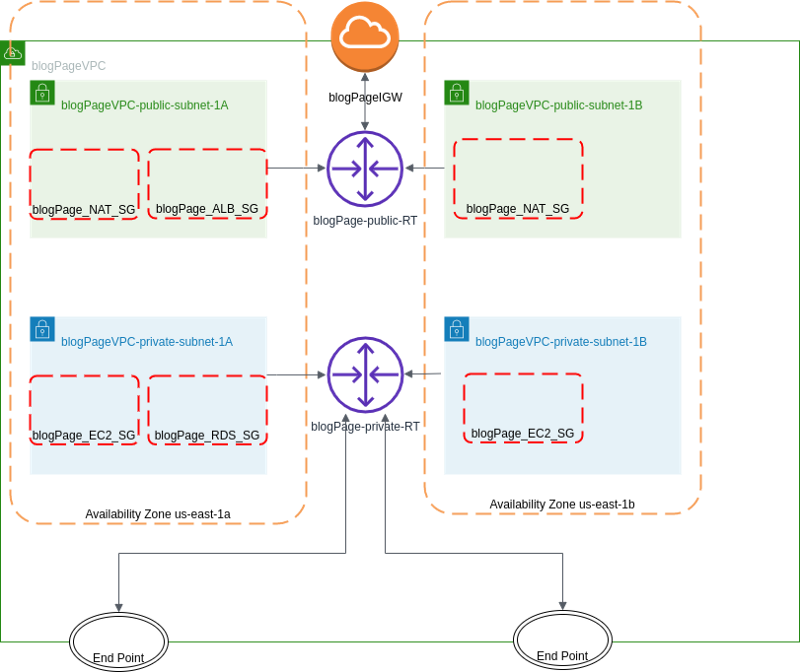
The purpose of making subnets private is to act as a firewall and to put them in a place that cannot be reached from the outside world. Just like keeping your most valuable pieces in the background in a chess game, creating a protection shield around them to protect them from outside attacks. Then we need route tables to enable communication between subnets. Route tables are navigation devices within the VPC.

When resources in subnets want to reach any port, external world or any resource internally, Route tables take them to the place they want to reach or block them. I created two Route tables. If you set a rule from the Route table and set this incoming traffic to the Internet Gateway, then this route table is called a public Route table. The subnets you attach to this Route table are also called public subnets. If you have not authorized the subnets you have connected to the route table to open to the outside world, these Route tables are called private Route tables. The subnets you attach to these Route tables also become private subnets. I have associated my private subnets with the private Route table and my public subnets with the public Route table.

Then I created the Internet Gateway named `blogPageIGW`, my resources can access the internet. Then attached the internet gateway to the created VPC `blogPageVPC`.

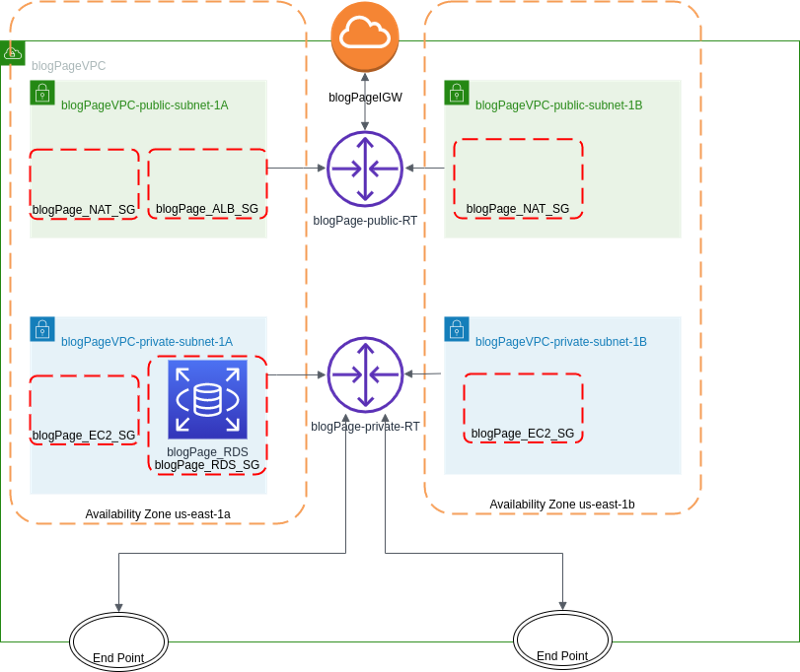
I created Endpoint for S3 Gateway. I have associated this Endpoint with the private route tables that I will also use.

#### Step 2: Create Security Groups (ALB — -> EC2 — -> RDS)



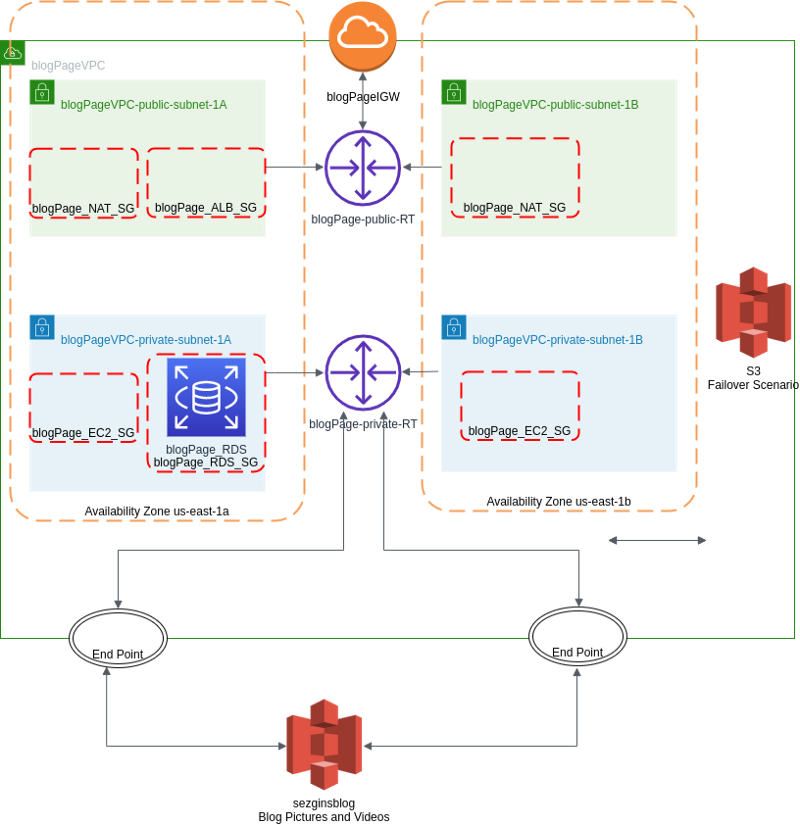
I created security groups before creating the instances. I have created separate security groups for ALB, EC2 and RDS. Firstly I created security group for ALB. The point you need to pay attention to while creating the security group here is the port of the listener and target groups. Django application will broadcast on port 80 and port 443. We will even set 443 and forward 443 to 80 and create a secure connection. Then I created the RDS security group. I added EC2 security group and SSH to inbound rules. I created a security group for the NAT instance. I opened HTTP, HTTPS and SSH ports as inbound rule.

#### Step 3: Create RDS



Before creating RDS, I created the DB security group inside the private subnet. I created my RDS as MySQL version 8.0.20. It is important to save the master username and password created here so that RDS will be connected with these credentials.

#### Step 4: Create two S3 Buckets and set one of these as a static website



One of the S3s will be used as a failover scenario and the other will be used for storing videos and photos. S3 Gateway Endpoint was established because the S3, where videos and photos are stored, is located outside the subnets and this traffic is not wanted to be exposed to the outside world.

#### Step 5: Download or clone project definition from repo on GitHub

I cloned the repository written by the developers on GitHub to my local.

#### Step 6: Prepare your GitHub repository

I created a private project repository on my GitHub and cloned it on your local. Copied all files and folders which are downloaded from GitHub repo under this folder. Committed and pushed them on your private GitHub repo.

#### Step 7: Prepare a userdata to be utilized in Launch Template

First of all, I created [userdata](https://github.com/sezginerdem/p12-Django-Blog-Page-App-on-AWS/blob/main/userdata.sh) to create Launch Templates. I made the updates so that the Instances can work in this userdata, I gave access to my GitHub account and I downloaded the necessary packages for the program to work. Then I pushed it to my GitHub repo.

```bash

#!/bin/bash

apt-get update -y

apt-get install git -y

apt-get install python3 -y

cd /home/ubuntu/

TOKEN="ghp\_HumpM7osFGfOWJNEQsbaM9YdiljNJ51N46bE"

git clone https://$TOKEN@[github.com/sezginerdem/p12-Django-Blog-Page-App-on-AWS.git](https://github.com/sezginerdem/p12-Django-Blog-Page-App-on-AWS.git)

cd /home/ubuntu/[p12-Django-Blog-Page-App-on-AWS](https://github.com/sezginerdem/p12-Django-Blog-Page-App-on-AWS.git)

apt install python3-pip -y

apt-get install python3.7-dev default-libmysqlclient-dev -y

pip3 install -r requirements.txt

cd /home/ubuntu/[p12-Django-Blog-Page-App-on-AWS](https://github.com/sezginerdem/p12-Django-Blog-Page-App-on-AWS.git)/src

python3 manage.py collectstatic --noinput

python3 manage.py makemigrations

python3 manage.py migrate

python3 manage.py runserver 0.0.0.0:80

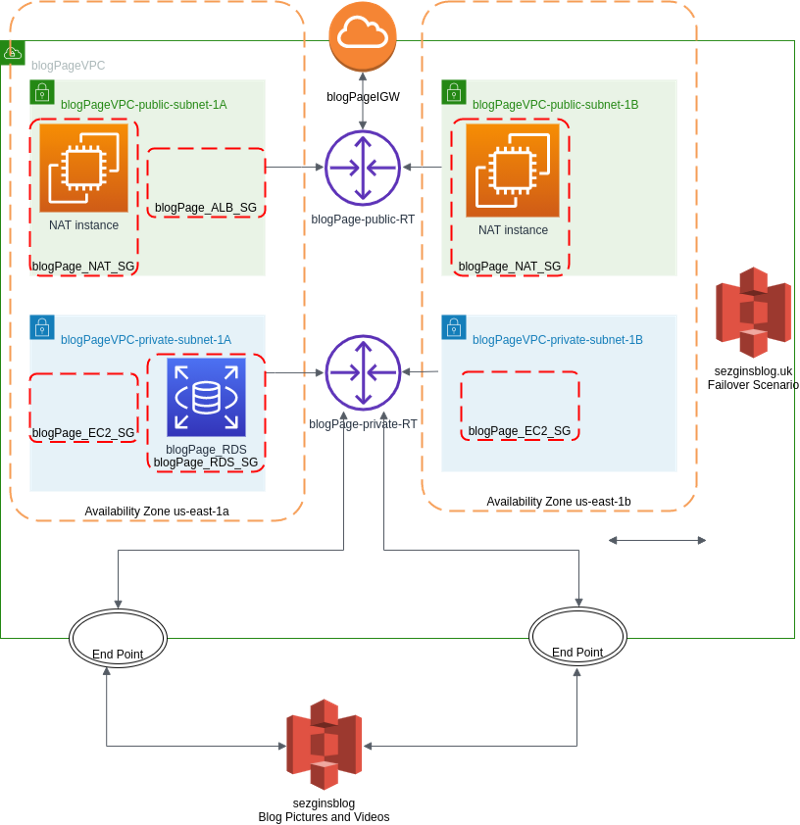
```

#### Step 8: Write RDS, S3 in the settings file given by the Fullstack Developer team

I assigned the name of S3 to the variable named AWS\_STORAGE\_BUCKET\_NAME in the settings.py file. I assigned the region name to AWS\_S3\_REGION\_NAME variable.

I have assigned my PASSWORD, which I have determined for RDS, into the .env file. I have assigned the information about the database in the DATABASES section of the settings.py file. I added the database information that I created in RDS here. It is important to enter this information correctly, as the connection to the database will be established through the information here.

#### Step 9: Create NAT Instance in Public Subnet



AutoScaling starts instances via Launch Template. In order to install patches and update the instances in the public subnet, I need to access them. For this, I created an instance on the public subnet, which we call a bastion host or jump box. This instance allowed me to access private instances. A NAT instance must be created for an internal instance to communicate with the outside. NAT Instance is a special type of instance. I created these instances via `amzn-ami-vpc-nat-hvm-2018.03.0.20181116-x86\_64-ebs`in the public subnet. After creating a NAT instance, the instance in the private subnet reaches the NAT instance in the public subnet via the private Route table, thus providing access to the outside world through the NAT instance. Besides, another feature of NAT instances can be used as a bastion host. Because these are also instances. The point to be noted here is that it is necessary to open SSH and HTTP and HTTPS ports everywhere. Because it can be reached as far as the open ports of the NAT instance can go. I created the NAT instance from the AMIs specially offered by AWS, so I did not have to do any additional configuration.

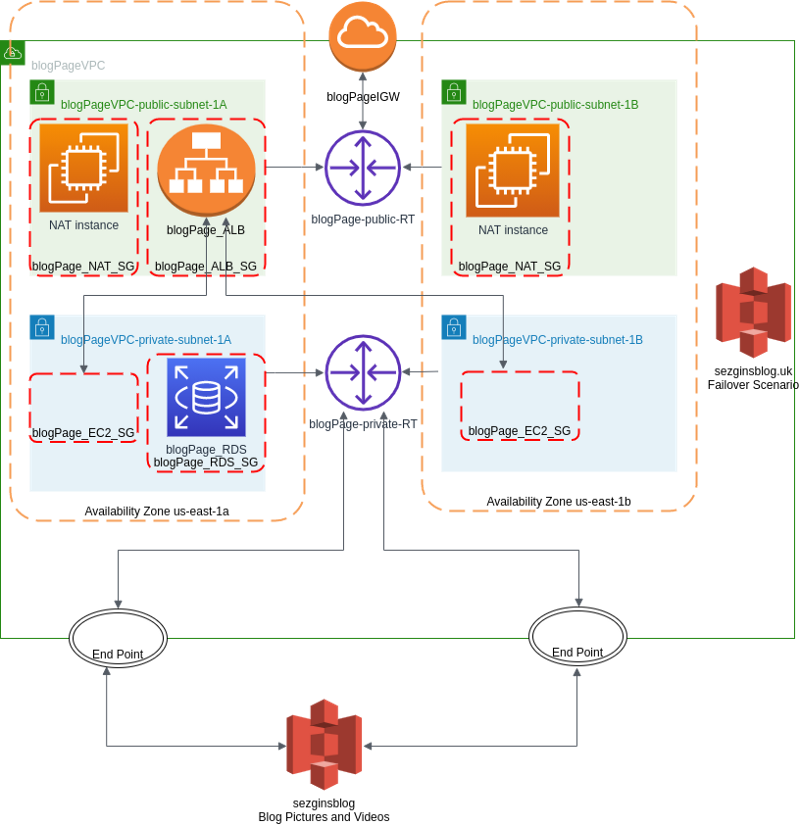
#### Step 10: Create Launch Template and IAM role for it

In order to create an instance from the launch template, it is necessary to assign a role. I created a role in AWS for this. Here I have assigned S3 full access authority to EC2. I created my Launch Template and wrote the userdata in my Launch Template.

#### Step 11: Create certification for secure connection

I went to the certification manager console and requested a certificate from AWS. I got a certificate for a secure connection. It took nearly one hour to be activated.

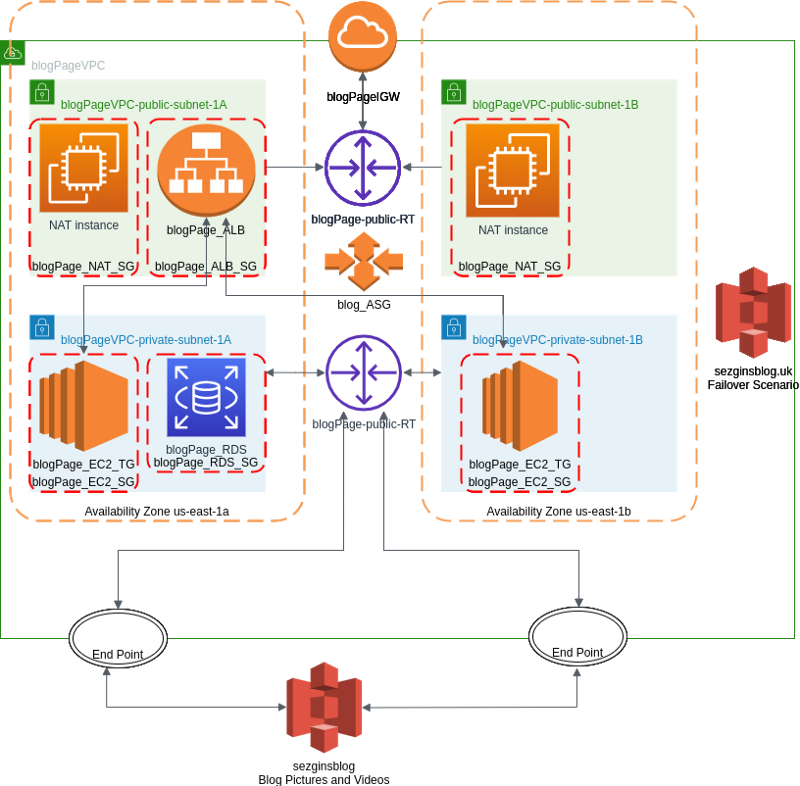
#### Step 12: Create ALB and Target Group



EC2s will work on the ELB. An application load balancer based load balancer was used as the ELB. The load balancer running on Layer 7 will enable all applications running on EC2s to be exposed to the outside world. The most important benefit of this ELB is that it distributes the traffic, provides it from a single source, and also enables the client from the outside world to reach the site without any disruption.

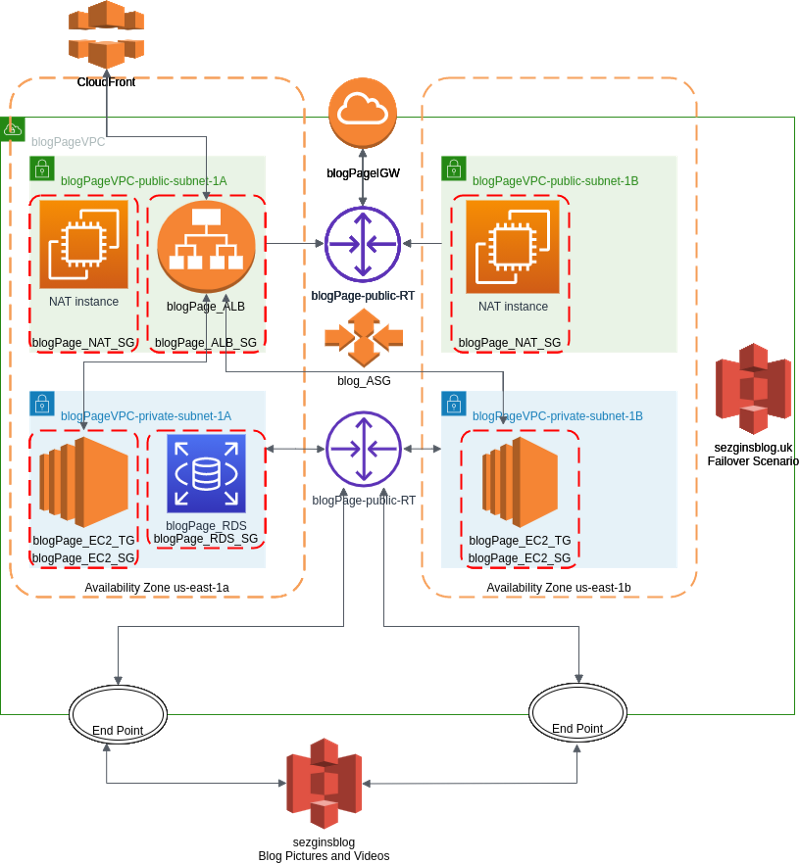
I associated the ALB with the certificate I received when creating the ALB. I set the instances as the target group. After creating the ALB, I forwarded the HTTP connection to HTTPS, updated it by doing the redirect rule.

#### Step 13: Create AutoScaling Group with Launch Template



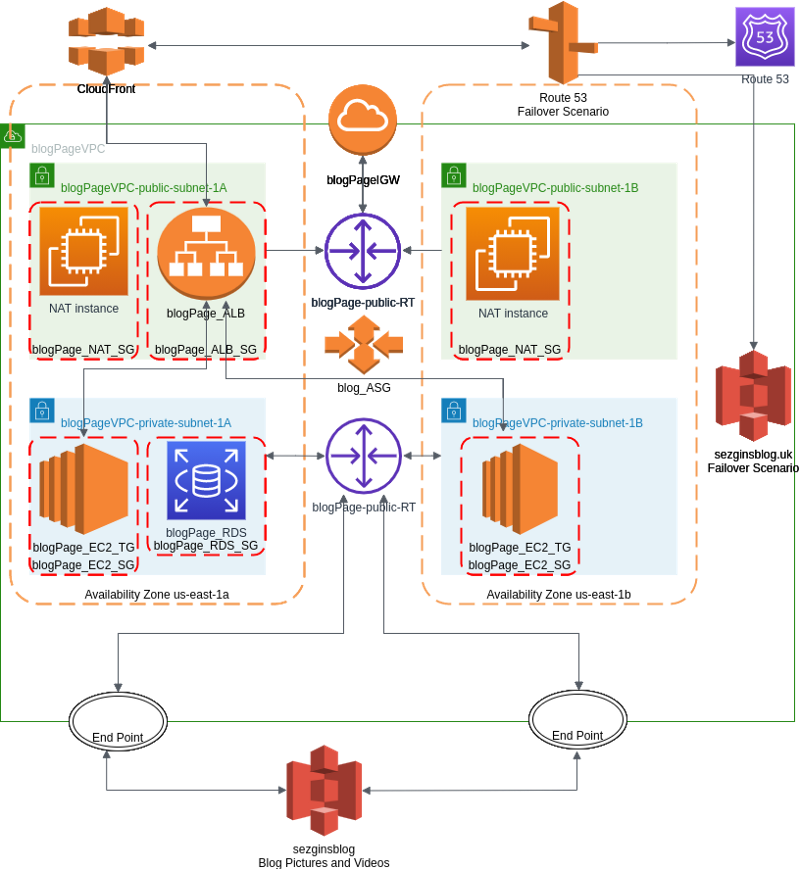
I created an Auto Scaling group. The target group I created for ALB is followed by the Auto Scaling group. If you can see that the application is running at this point, it means that your application has been deployed without any problems.

#### Step 14: Create CloudFront in front of ALB



I created CloudFront as a cash service for ELB in order to prevent videos and photos from being taken from S3 every time, to make my application work faster and to make it cost effective.

#### Step 15: Create Route 53 with Failover settings



Added failover scenario while configuring Route53. Failover scenarios follow two endpoints. One of the endpoints is generally static websites. The other is your dynamically running website. If something happens to my dynamically running website, it will redirect my traffic to my secondary route. Thus, in case the clients cannot access my website, they will have access to my static website.

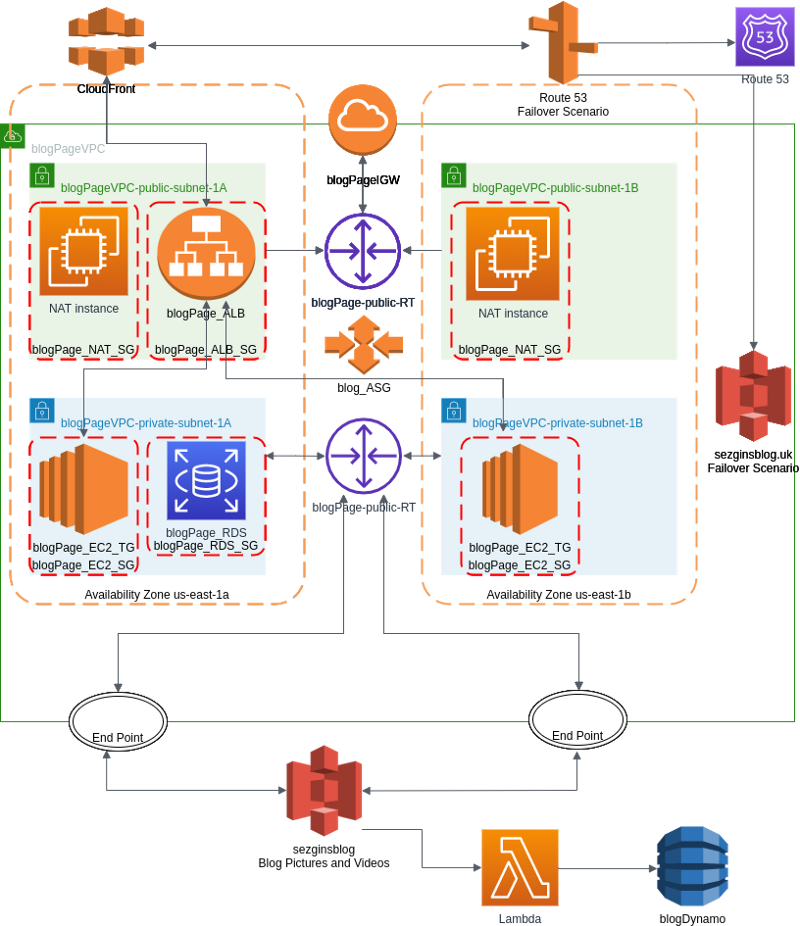
#### Step 16: Create DynamoDB Table

One of the demands of the project is to keep a list of all transactions made in S3. Any object added or deleted here is set to generate an event. This event will trigger the Lambda function and has been added to DynamoDB as a list. This list is updated whenever something is added or removed from this list. This list is kept up to date in DynamoDB.

#### Step 17–18: Create Lambda function

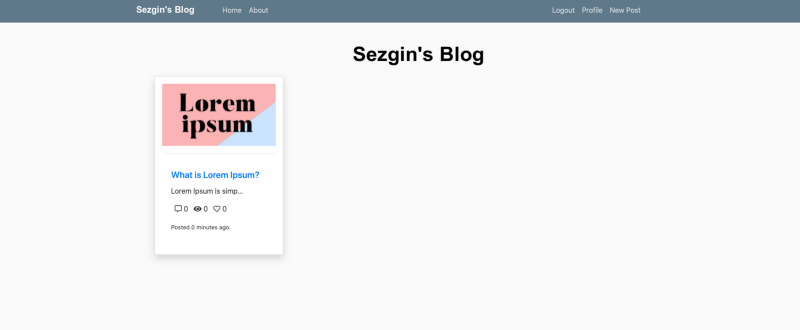
The Lambda function I will create must have permission to access S3. It also needs to talk to DynamoDB and give some permissions about the network. For the purpose of assigning these privileges, I created an IAM role and assigned this role to Lambda, which defines S3, DynamoDB and network full access privileges. Then I created Lambda function.

#### Step 17–18: Create S3 Event and set it as trigger for Lambda Function



I created an event from the event notification tab from properties in S3. I set these two events as adding and deleting an object in S3. After I created my Lambda function, I created events in S3 and enabled these events to trigger Lambda. I added my lambda code to the function. Thus, my Lambda function also became functional. Thus, my list in DynamoDB started to be kept up to date.

My blog site can also be seen in its final form in a fully deployed state.



### References:

[**sezginerdem/p12-Django-Blog-Page-App-on-AWS**  
*The Clarusway Blog Page Application aims to deploy blog application as a web application written Django Framework on…*github.com](https://github.com/sezginerdem/p12-Django-Blog-Page-App-on-AWS.git)

[](https://faun.to/i9Pt9)

Join FAUN: [**Website**](https://faun.to/i9Pt9) 💻|[**Podcast**](https://faun.dev/podcast) 🎙️|[**Twitter**](https://twitter.com/joinfaun) 🐦|[**Facebook**](https://www.facebook.com/faun.dev/) 👥|[**Instagram**](https://instagram.com/fauncommunity/) 📷|[**Facebook Group**](https://www.facebook.com/groups/364904580892967/) 🗣️|[**Linkedin Group**](https://www.linkedin.com/company/faundev) 💬| [**Slack**](https://faun.dev/chat) 📱|[**Cloud Native** **News**](https://thechief.io/) 📰|[**More**](https://linktr.ee/faun.dev/).

If this post was helpful, please click the clap 👏 button below a few times to show your support for the author 👇