

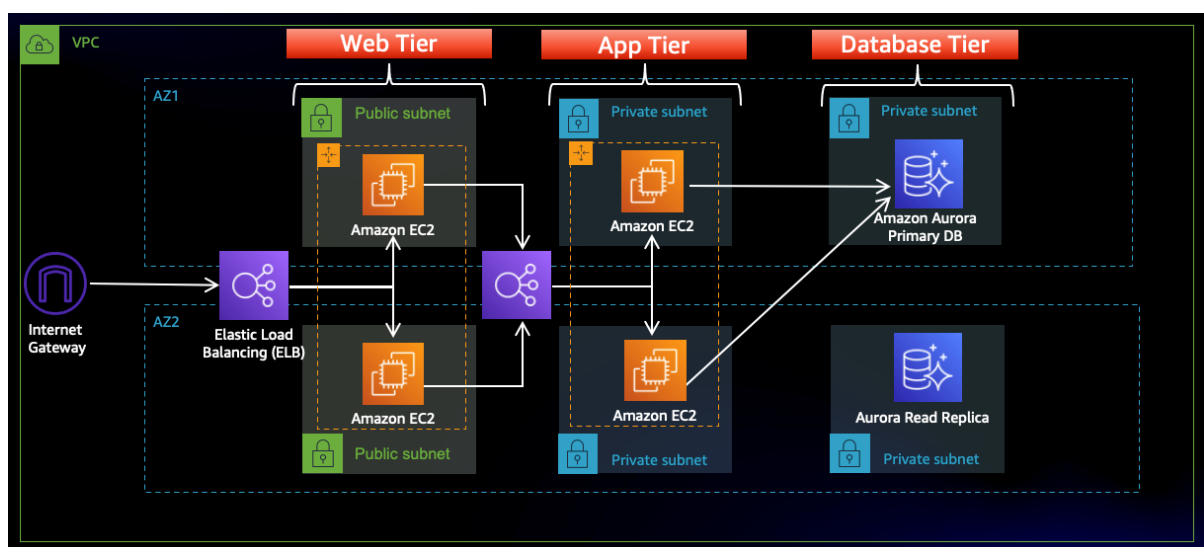
Project: AWS Three Tier Web Architecture

By,

Karthik NP

DevOps Engineer

Architecture overview:



In this architecture, a public-facing Application Load Balancer forwards client traffic to our web tier EC2 instances. The web tier is running Nginx webserver that are configured to serve a React.js website and redirects our API calls to the application tier's internal facing load balancer. The internal facing load balancer then forwards that traffic to the application tier, which is written in Node.js. The application tier manipulates data in an Aurora MySQL multi-AZ database and returns it to our web tier. Load balancing, health checks and autoscaling groups are created at each layer to maintain the availability of this architecture.

Step.1: Downloading the Code from GitHub

I downloaded the code from AWS repo into my local environment by running the command below.

```
$ git clone https://github.com/aws-samples/aws-three-tier-web-architecture-workshop.git
```

```
Windows PowerShell
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\hp> mkdir 3tier

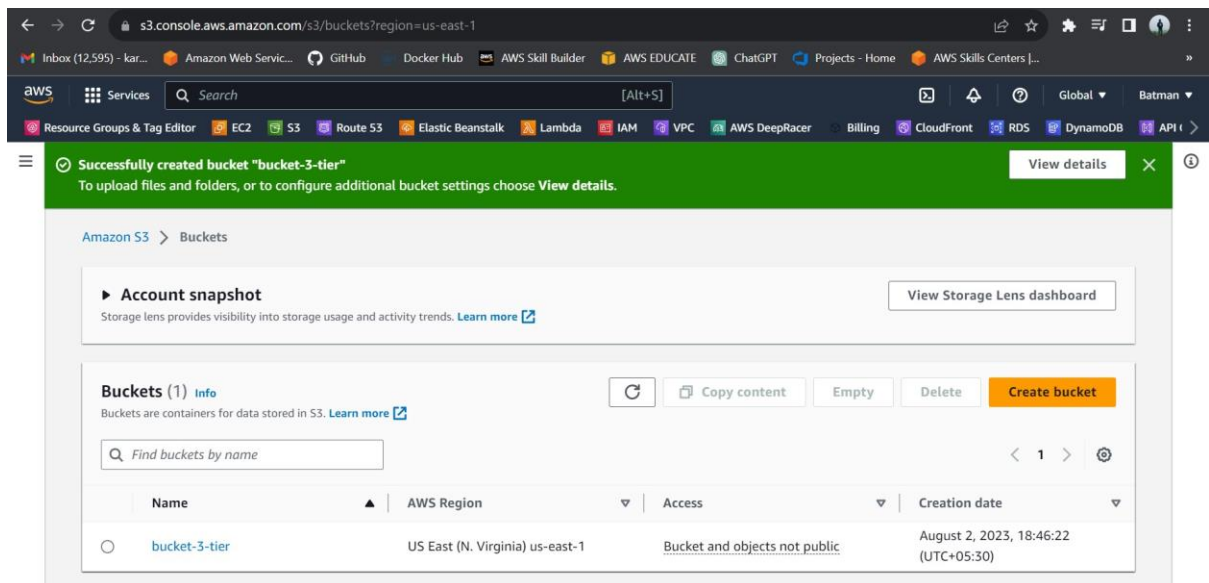
Directory: C:\Users\hp

Mode                LastWriteTime         Length Name
----                -
d-----          02-08-2023      18:42         3tier

PS C:\Users\hp> cd 3tier
PS C:\Users\hp\3tier> git clone https://github.com/aws-samples/aws-three-tier-web-architecture-workshop.git
>>
>>
Cloning into 'aws-three-tier-web-architecture-workshop'...
remote: Enumerating objects: 133, done.
remote: Counting objects: 100% (133/133), done.
remote: Compressing objects: 100% (102/102), done.
Receiving objects: 65% (87/133)remote: Total 133 (delta 55), reused 90 (delta 24), pack-reused 0
Receiving objects: 100% (133/133), 224.38 KiB | 2.06 MiB/s, done.
Resolving deltas: 100% (55/55), done.
PS C:\Users\hp\3tier>
```

Step.2: S3 Bucket Creation

Then I created an S3 bucket, with all the defaults configurations as in. This bucket is where I will upload my code later.

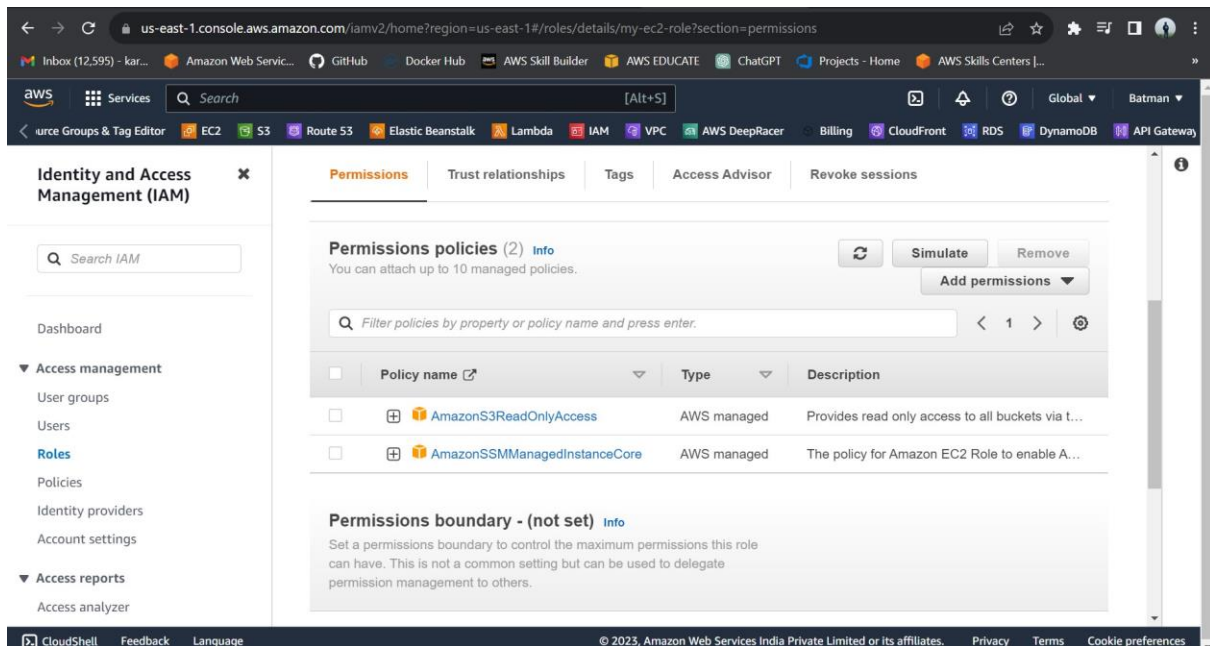


Step.3: IAM EC2 Instance Role Creation

- Created an EC2 role including the following AWS managed policies. These policies will allow our instances to download our code from S3 and use Systems Manager Session Manager to securely connect to our instances without SSH keys through the AWS console.

1. AmazonSSMManagedInstanceCore

2. AmazonS3ReadOnlyAccess



Step.4: Networking and Security

I built out the VPC networking components as well as security groups that will add a layer of protection around our EC2 instances, Aurora databases, and Elastic Load Balancers.

- Created an isolated network using the following components:
 - VPC
 - Subnets
 - Route Tables
 - Internet Gateway
 - NAT gateway
 - Security Groups

VPC Creation:

I have chosen a CIDR range that will allow us to create at least 6 subnets.

Subnet Creation:

We will need **six** subnets across **two** availability zones. Each subnet in one availability zone will correspond to one layer of our three-tier architecture.

Note: Our CIDR range for the subnets will be subsets of our VPC CIDR range.

Internet Connectivity:

- Internet Gateway

In order to give the public subnets in my VPC internet access, I created and attached an Internet Gateway and attached it to my VPC.

- NAT Gateway

In order for our instances in the app layer private subnet to be able to access the internet they will need to go through a NAT Gateway. For high availability, I deployed one NAT gateway in each of our **public** subnets and then allocated an Elastic IP for each.

Routing Configuration:

I created one route table for the web-layer public subnets. Added a route that directs traffic from the VPC to the internet gateway.

Then I changed the Subnet Associations by selecting the two web-layer public subnets I created earlier and created 2 more route tables, one for each app layer private subnet in each availability zone. These route tables will route app layer traffic destined for outside the VPC to the NAT gateway in the respective availability zone. Then add the subnet associations for each of the app layer private subnets.

Security Groups:

Security groups will tighten the rules around which traffic will be allowed to our Elastic Load Balancers and EC2 instances.

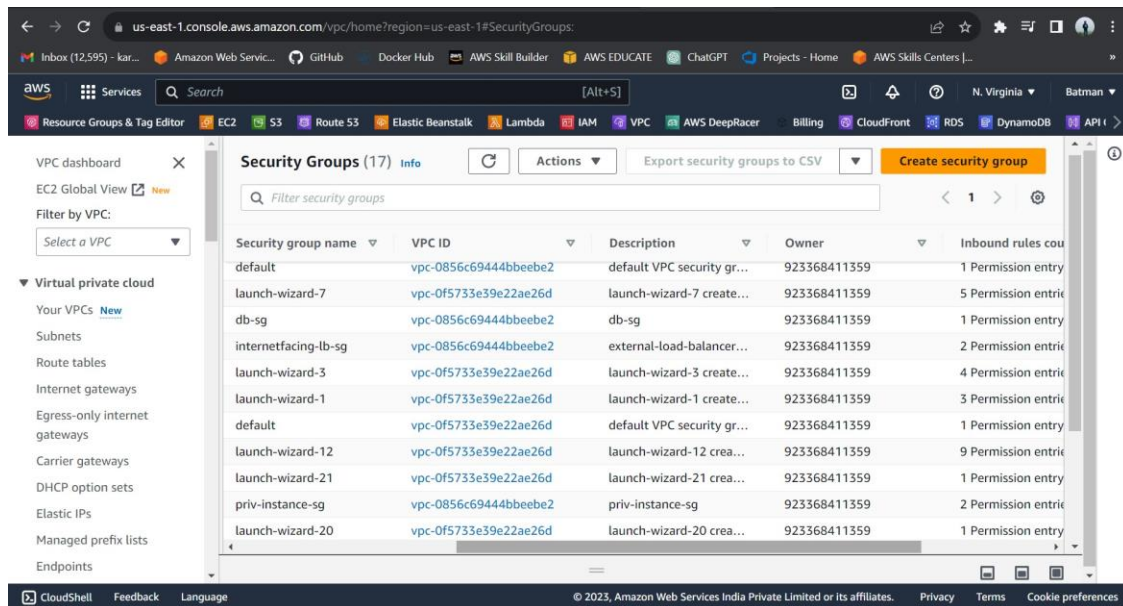
The first security group I created is for the public, **internet facing** load balancer. Then added an inbound rule to allow **HTTP** type traffic for my **IP**.

The second security group I created is for the public instances in the web tier, then added an inbound rule that allows **HTTP** traffic from my internet facing load balancer security group. This will allow traffic from my public facing load balancer to hit our instances. Then, added an additional rule that will allow HTTP type traffic for my IP.

The third security group will be for my internal load balancer. Then added an inbound rule that allows **HTTP** type traffic from my public instance security group. This will allow traffic from our web tier instances to hit my internal load balancer.

The fourth security group I created is for our private instances. Then added an inbound rule that will allow **TCP** type traffic on port **4000** from the **internal load balancer security group**. This is the port our app tier application is running on and allows our internal load balancer to forward traffic on this port to our private instances.

The fifth security group I created protects our private database instances. I added an inbound rule that will allow traffic from the **private instance security group** to the **MySQL/Aurora port (3306)**.



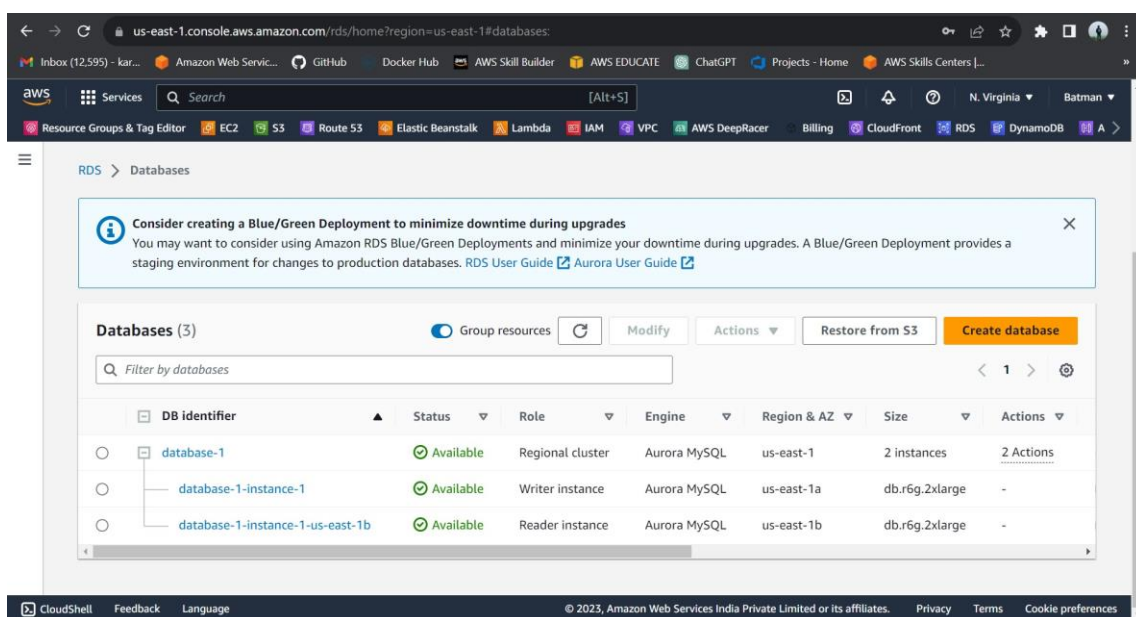
Step.5: Database Deployment

Subnet Groups:

Added the subnets that created in each availability zone specifically for our database layer.

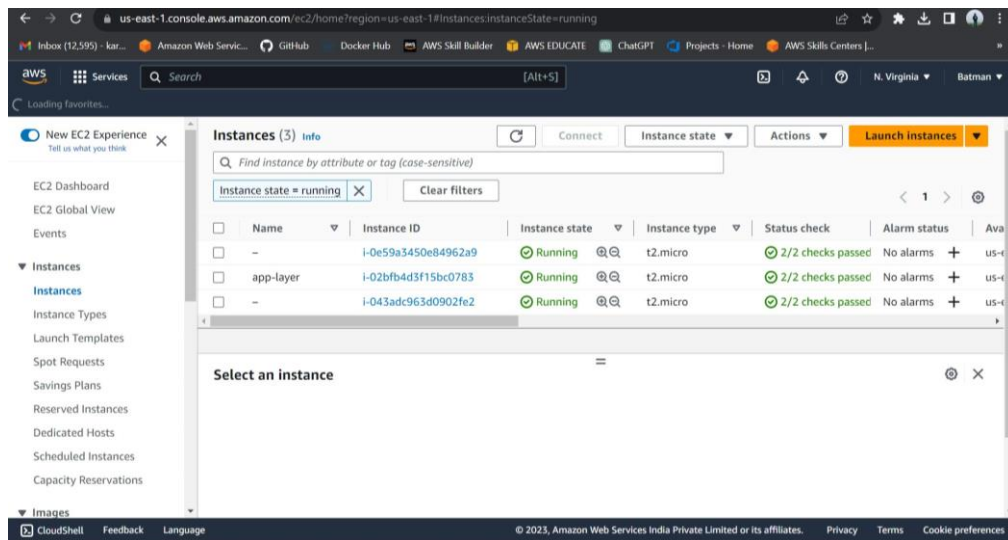
Database Deployment:

I selected a **Standard create** for this **MySQL-Compatible Amazon Aurora** database. I chose **Dev/Test** as a template since this isn't being used for production at the moment. I set a username and password to access our database. Next, under Availability and durability, I changed the option to create an Aurora Replica or reader node in a different availability zone. I set the security group that created for the database layer.



Step.6: App Tier Instance Deployment

I created an EC2 instance for our app layer. The app layer consists of a Node.js application that will run on port 4000. I chose **Amazon Linux 2 AMI** with **t.2 micro** instance type and select the correct VPC, subnet, and IAM role. Since this is the app layer, I used one of the private subnets. Then selected a security group created for our private app layer instances. Since I'm using Systems Manager Session Manager to connect to the instance, proceeded without a keypair.

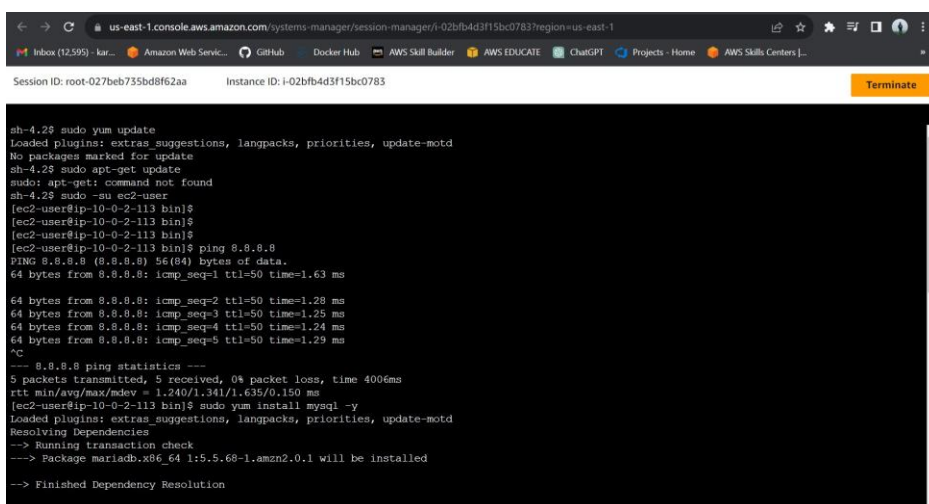


- I connected my instance through the Session Manager. Switched to ec2-user by executing the following command:

```
$ sudo -su ec2-user
```

- To make sure that I'm able to reach the internet via our NAT gateways:

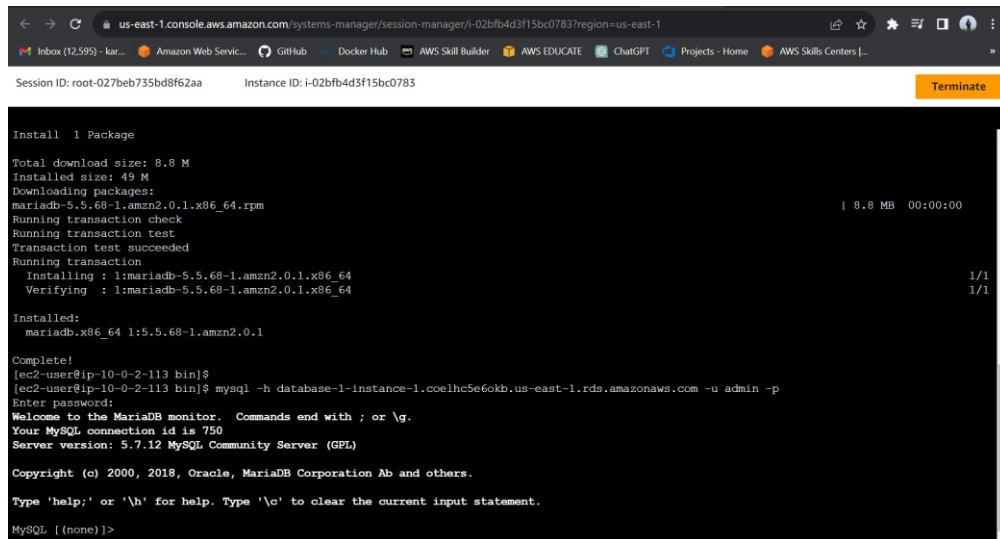
```
$ ping 8.8.8.8
```



Configuring the Database:

- Download the MySQL CLI:

\$ **sudo yum install** mysql -y



The screenshot shows a terminal window within an AWS Systems Manager session. The terminal output displays the installation of the MySQL CLI package. It shows the download size (9.8 M), installed size (49 M), and the package name (mariadb-5.5.68-1.amzn2.0.1.x86_64.rpm). The transaction check and test are successful. The installation is complete, and the user is prompted to enter a password. The terminal output shows the user entering a password and the MySQL CLI prompt (mysql) appearing. The user then enters the command 'mysql -h database-1-instance-1.coe1hc5e6okb.us-east-1.rds.amazonaws.com -u admin -p' and the prompt returns to the MySQL CLI prompt (mysql [(none)]>).

- Initiating my DB connection with my Aurora RDS writer endpoint. In the following command, I replaced the RDS writer endpoint and the username, and then executed.

\$ **mysql -h** CHANGE-TO-YOUR-RDS-ENDPOINT **-u** CHANGE-TO-**USER**-NAME -p

\$ **mysql -h** database-1-instance-1.coe1hc5e6okb.us-east-1.rds.amazonaws.com **-u** admin -p

Then I entered the password in the prompt and connected to my database.

- Then created a database called **webappdb** with the following command using the MySQL CLI:

CREATE DATABASE webappdb;

- To verify that it was created correctly:

SHOW DATABASES;

- Then created a data table by first navigating to the database we just created:

USE webappdb;

- Then, created the following **transactions** table by executing this create table command:

```
CREATE TABLE IF NOT EXISTS transactions(id INT NOT NULL  
AUTO_INCREMENT, amount DECIMAL(10,2), description  
VARCHAR(100), PRIMARY KEY(id));
```

- To verify the table was created:

```
SHOW TABLES;
```

- Inserted some data into table for use/testing later:

```
INSERT INTO transactions (amount,description) VALUES ('400','groceries');
```

- To verify that my data was added:

```
SELECT * FROM transactions;
```

Exited from the MySQL client.

Configuring the App Instance:

I updated the database credentials for the app tier by opening the **application-code/app-tier/DbConfig.js** file from the GitHub repo in VS Code. Filled the strings for the hostname, user, password and database with my credentials, the **writer endpoint** of my database as the hostname, and **webappdb** for the database and saved. Then uploaded the **app-tier** folder to my S3 bucket.

Now I installed all of the necessary components to run our backend application.

- Started by installing NVM (node version manager).

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash
```

```
$ source ~/.bashrc
```

- Next, I installed a compatible version of Node.js and made sure it's being used

```
$ nvm install 16
```

```
$ nvm use 16
```

- PM2 is a daemon process manager that will keep our node.js app running when we exit the instance or if it is rebooted. So I installed that as well.

```
$ npm install -g pm2
```

- Now I downloaded our code from our S3 bucket onto our instance.

```
$ cd ~/
```

```
$ aws s3 cp s3://bucket-3-tier/app-tier/ app-tier --recursive
```


- Entered into the app directory, installed dependencies, and started the app with pm2.

```
$ cd ~/app-tier
```

```
$ npm install
```

```
$ pm2 start index.js
```

[illegible]

- To make sure the app is running correctly:

```
$ pm2 list
```

- pm2 is just making sure our app stays running when I leave the SSM session. However, if the server is interrupted for some reason, we still want the app to start and keep running. This is also important for the AMI we will create later:

```
$ pm2 startup
```

- To setup the Startup Script, I pasted the command I got that started with

```
$ sudo env PATH=$PATH:/home/ec2-user....
```

- To save the current list of node processes:

\$ pm2 save

Testing App Tier:

- To hit out health check endpoint:

```
$ curl http://localhost:4000/health
```

- The response should look like this:

"This is the health check"

- Then I tested our database connection by hitting the following endpoint locally:

curl http://localhost:4000/transaction

I got the response containing the test data I added earlier:

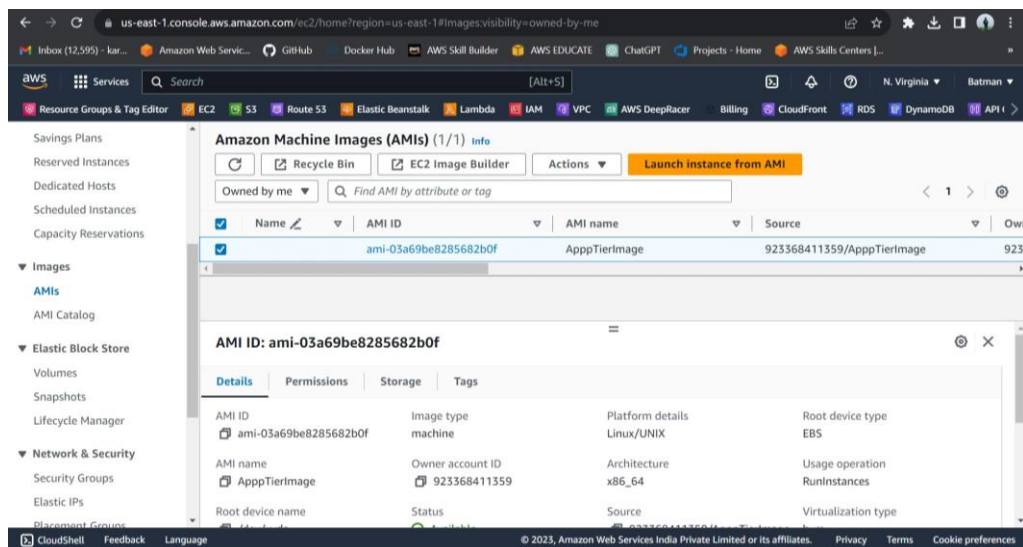
```
{ "result": [{ "id": 1, "amount": 400, "description": "groceries" }, { "id": 2, "amount": 100, "description": "class" }, { "id": 3, "amount": 200, "description": "other groceries" }, { "id": 4, "amount": 10, "description": "brownies" } ] }
```

Step.7: Internal Load Balancing and Auto Scaling

I created an Amazon Machine Image (AMI) of the app tier instance I created, and use that to set up autoscaling with a load balancer in order to make this tier highly available.

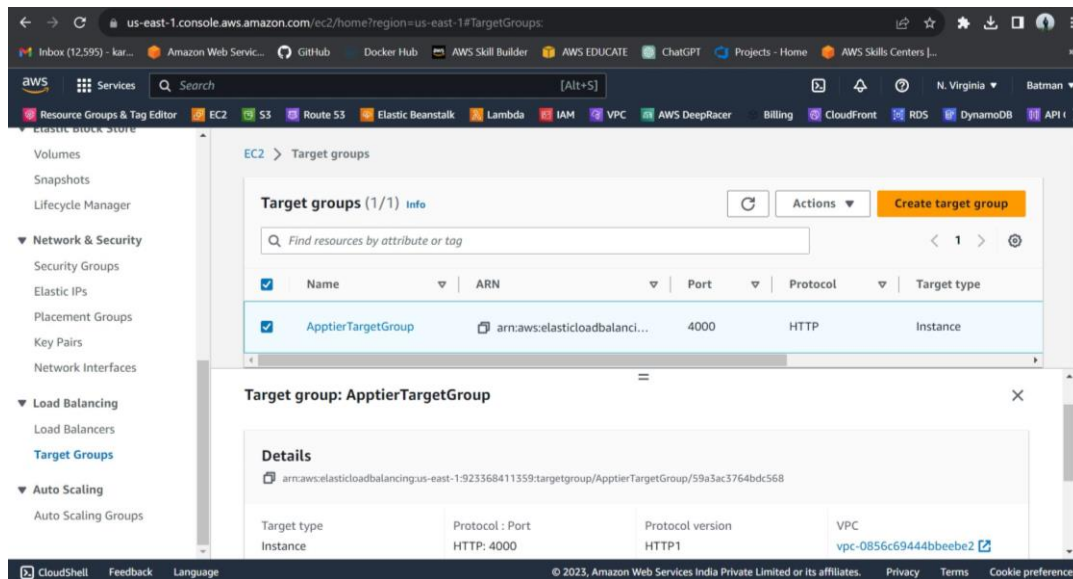
App Tier AMI:

I created an Image of the app-tier instance.



Target Group:

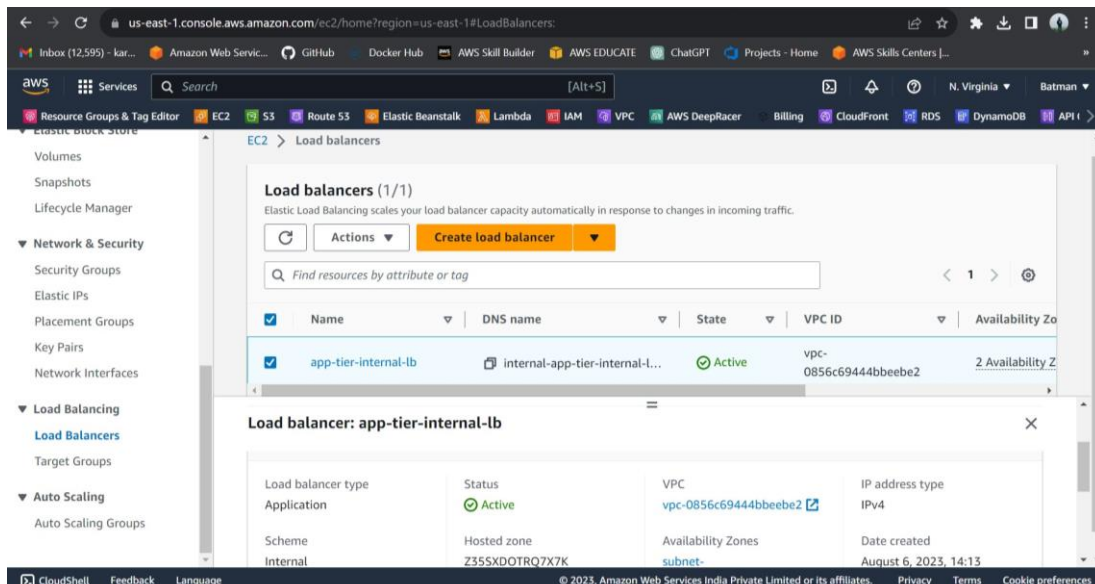
Then I create a target group to use with the load balancer. The purpose of forming this target group is to use with our load balancer so it may balance traffic across our private app tier instances. Then I set the protocol to **HTTP** and the port to 4000, this is the port our Node.js app is running on. Changed the health check path to be **/health**. This is the health check endpoint of our app. I did not register any targets for now.



Internal Load Balancer:

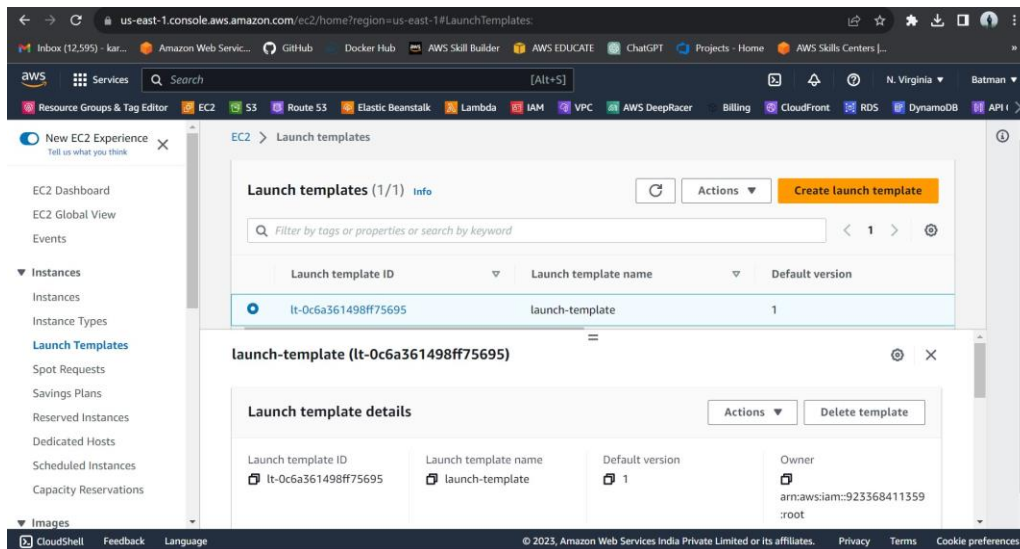
I created an Application Load Balancer for our **HTTP** traffic and selected **internal** since this one will not be public facing, but rather it will route traffic from our web tier to the app tier. Then selected the VPC and private subnets.

Then selected the security group I created for this internal ALB. Now, this ALB will be listening for HTTP traffic on port 80. It will be forwarding the traffic to our target group that I just created.



Launch Template: I created a Launch template with the AMI I created earlier with the app tier created. For Key pair and Network Settings, I didn't include it in the template. Since we don't need a key pair to access our instances and we'll be setting the network information in

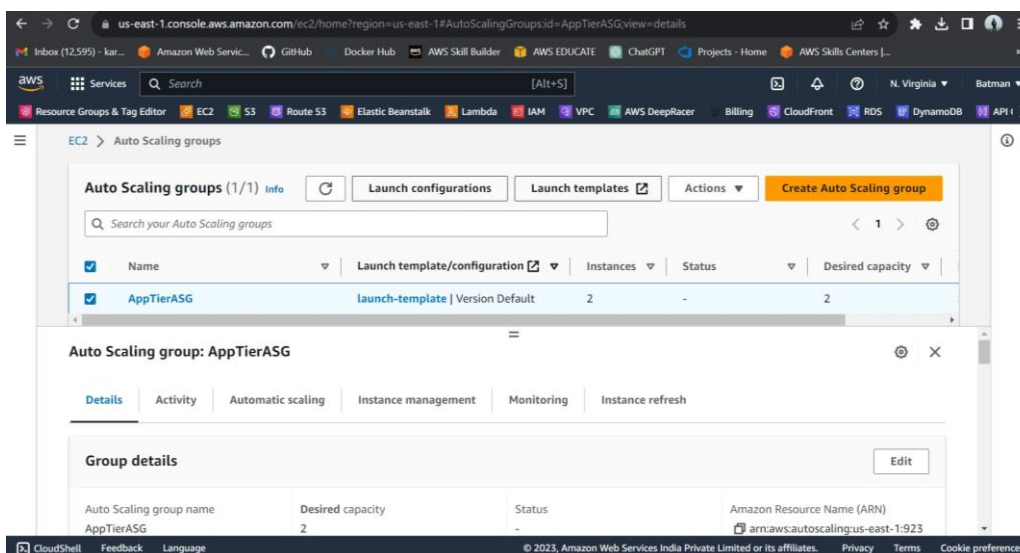
the autoscaling group. Then I used the security group for our app tier, and the same IAM role I have been using for our EC2 instances.



Auto Scaling:

I now created the Auto Scaling Group for our app instances by setting my VPC, and the private instance subnets for the app tier. Then attached this Auto Scaling Group to the Load Balancer I just created by selecting the existing load balancer's target group. I set desired, minimum and maximum capacity to 2.

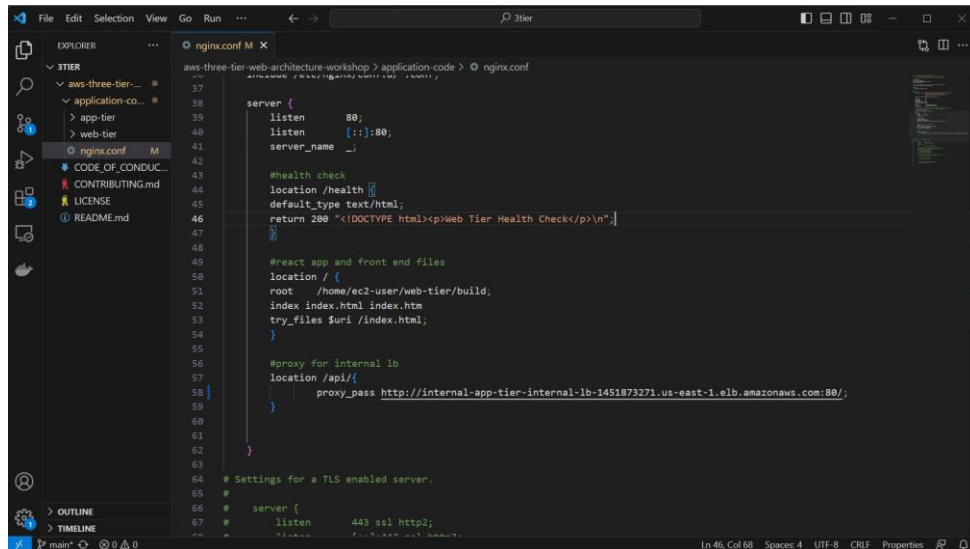
I now have our internal load balancer and autoscaling group configured correctly.



Step.8: Web Tier Instance Deployment: I deployed an EC2 instance for the web tier and made all necessary software configurations for the NGINX web server and React.js website.

Updating the Config File:

In the line **58** of the **application-code/nginx.conf** file I replaced [INTERNAL-LOADBALANCER-DNS] with my internal load balancer's DNS entry. Then, uploaded this file and the **application-code/web-tier** folder to my S3 bucket.



```
server {
    listen      80;
    listen      [::]:80;
    server_name _;

    #health check
    location /health {
        default_type text/html;
        return 200 "<DOCTYPE html><p>Web Tier Health Check</p>\n";
    }

    #react app and front end files
    location / {
        root    /home/ec2-user/web-tier/build;
        index   index.html index.htm;
        try_files $uri /index.html;
    }

    #proxy for internal lb
    location /api/{
        proxy_pass http://internal-app-tier-internal-lb-1451873271.us-east-1.elb.amazonaws.com:80;
    }
}
```

Web Instance Deployment:

I placed this web-tier instance in one of our **public subnets**. Then selected the correct network components, security group, and IAM role. I enabled the, auto-assign a public ip. And then proceeded without a key pair for this instance.

Connecting to Instance:

```
$ sudo -su ec2-user
```

```
$ ping 8.8.8.8
```

Configure Web Instance:

- I installed all of the necessary components needed to run our front-end application. Again, started by installing NVM and node :

```
$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash
```

```
$ source ~/.bashrc
```

```
$ nvm install 16
```

```
$ nvm use 16
```

```
us-east-1.console.aws.amazon.com/systems-manager/session-manager/i-0b5f6b38936f8791b?region=us-east-1#
Inbox (12,595) - kar... Amazon Web Servic... GitHub Docker Hub AWS Skill Builder AWS EDUCATE ChatGPT Projects - Home AWS Skills Centers ...
Session ID: root-0a37ed716f7ad589d Instance ID: i-0b5f6b38936f8791b Terminate

10 packets transmitted, 10 received, 0% packet loss, time 9013ms
rtt min/avg/max/mdev = 1.197/1.236/1.282/0.053 ms
[ec2-user@ip-10-0-0-138 bin]$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload  Total    Spent    Left  Speed
100 14926 100 14926    0     0  192k      0  --:--:-- --:--:-- --:--:-- 194k
=> Downloading nvm as script to '/home/ec2-user/.nvm'

=> Appending nvm source string to /home/ec2-user/.bashrc
=> Appending bash completion source string to /home/ec2-user/.bashrc
=> Close and reopen your terminal to start using nvm or run the following to use it now:

export NVM_DIR="$HOME/.nvm"
[ -s "$NVM_DIR/nvm.sh" ] && \. "$NVM_DIR/nvm.sh" # This loads nvm
[ -s "$NVM_DIR/bash_completion" ] && \. "$NVM_DIR/bash_completion" # This loads nvm bash_completion
[ec2-user@ip-10-0-0-138 bin]$ source ~/.bashrc
[ec2-user@ip-10-0-0-138 bin]$ nvm install 16
Downloading and installing node v16.20.1...
Downloading https://nodejs.org/dist/v16.20.1/node-v16.20.1-linux-x64.tar.xz...
##### 100.0%
Computing checksum with sha256sum
Checksums matched!
Now using node v16.20.1 (npm v8.19.4)
Creating default alias: default -> 16 (-> v16.20.1)
[ec2-user@ip-10-0-0-138 bin]$ nvm use 16
Now using node v16.20.1 (npm v8.19.4)
[ec2-user@ip-10-0-0-138 bin]$ cd
[ec2-user@ip-10-0-0-138 ~]$ ls
[ec2-user@ip-10-0-0-138 ~]$ nvm
```

- Now I downloaded our web tier code from my S3 bucket:

```
$ cd ~/
```

```
$ aws s3 cp s3://bucket-3-tier/web-tier/ web-tier --recursive
```

- Entered into the web-layer folder and created the build folder for the react app so I can serve our code:

```
$ cd ~/web-tier
```

```
$ npm install
```

```
$ npm run build
```

```
us-east-1.console.aws.amazon.com/systems-manager/session-manager/i-0b5f6b38936f8791b?region=us-east-1#
Inbox (12,595) - kar... Amazon Web Servic... GitHub Docker Hub AWS Skill Builder AWS EDUCATE ChatGPT Projects - Home AWS Skills Centers ...
Session ID: root-0a37ed716f7ad589d Instance ID: i-0b5f6b38936f8791b Terminate

declaring it in its dependencies. This is currently working because
"@babel/plugin-proposal-private-property-in-object" is already in your
node_modules folder for unrelated reasons, but it may break at any time.

babel-preset-react-app is part of the create-react-app project, which
is not maintained anymore. It is thus unlikely that this bug will
ever be fixed. Add "@babel/plugin-proposal-private-property-in-object" to
your devDependencies to work around this error. This will make this message
go away.

Compiled successfully.

File sizes after gzip:

 77.41 kB build/static/js/main.e5e4b28e.js
 1.79 kB build/static/js/787.1f63e066.chunk.js
 493 B build/static/css/main.b20b6ac4.css

The project was built assuming it is hosted at ./..
You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.

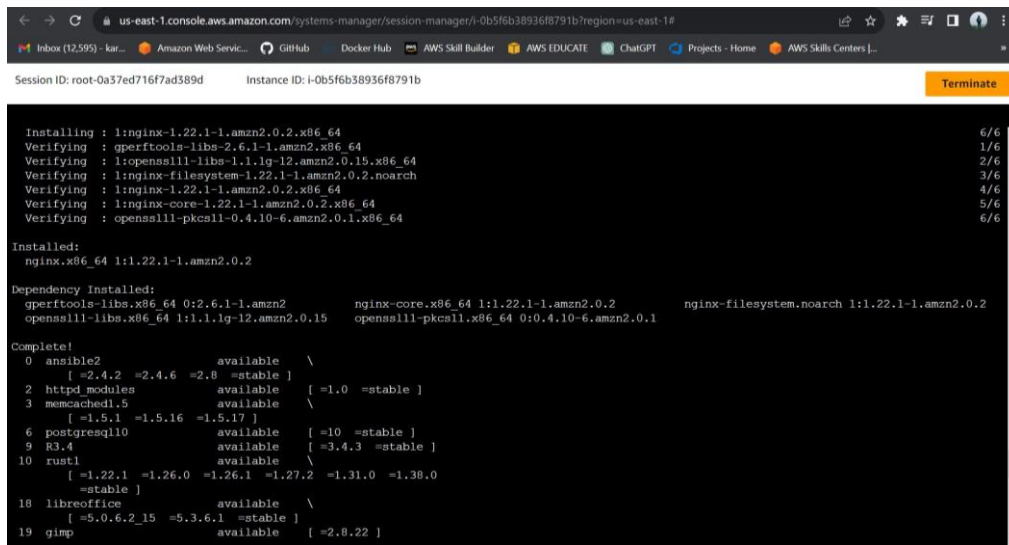
Find out more about deployment here:

 https://cra.link/deployment

[ec2-user@ip-10-0-0-138 web-tier]$
[ec2-user@ip-10-0-0-138 web-tier]$
```

NGINX can be used for different use cases like load balancing, content caching etc, but I'll be using it as a web server that I will configure to serve our application on port 80, as well as help direct our API calls to the internal load balancer.

\$ sudo amazon-linux-extras install nginx1 -y



```
Installing : 1:nginx-1.22.1-1.amzn2.0.2.x86_64 6/6
Verifying : gperftools-libs-2.6.1-1.amzn2.x86_64 1/6
Verifying : 1:openssl-libs-1.1.1g-12.amzn2.0.15.x86_64 2/6
Verifying : 1:nginx-filesystem-1.22.1-1.amzn2.0.2.noarch 3/6
Verifying : 1:nginx-1.22.1-1.amzn2.0.2.x86_64 4/6
Verifying : 1:nginx-core-1.22.1-1.amzn2.0.2.x86_64 5/6
Verifying : openssl-pkcs11-0.4.10-6.amzn2.0.1.x86_64 6/6

Installed:
  nginx.x86_64 1:1.22.1-1.amzn2.0.2

Dependency Installed:
  gperftools-libs.x86_64 0:2.6.1-1.amzn2      nginx-core.x86_64 1:1.22.1-1.amzn2.0.2      nginx-filesystem.noarch 1:1.22.1-1.amzn2.0.2
  openssl-libs.x86_64 1:1.1.1g-12.amzn2.0.15  openssl-pkcs11.x86_64 0:0.4.10-6.amzn2.0.1

Complete!
0  ansible2          available \
   [ =2.4.2 =2.4.6 =2.8 =stable ]
2  httpd_modules     available [ =1.0 =stable ]
3  memcached1.5      available \
   [ =1.5.1 =1.5.16 =1.5.17 ]
6  postgresql10      available [ =10 =stable ]
9  R3.4              available [ =3.4.3 =stable ]
10 rust1             available \
   [ =1.22.1 =1.26.0 =1.26.1 =1.27.2 =1.31.0 =1.38.0
   =stable ]
18 libreoffice        available \
   [ =5.0.6.2_15 =5.3.6.1 =stable ]
19 gimp               available [ =2.8.22 ]
```

- I now configured NGINX by going to the Nginx configuration file,

\$ cd /etc/nginx

\$ ls

- I deleted this nginx.conf file and used the one I uploaded to S3.

\$ sudo rm nginx.conf

\$ sudo aws s3 cp s3:// bucket-3-tier/web-tier /nginx.conf .

- Then, restarted Nginx:

\$ sudo service nginx restart

- To make sure Nginx has permission to access our files:

\$ chmod -R 755 /home/ec2-user

- And then to make sure the service starts on boot:

\$ chkconfig nginx on

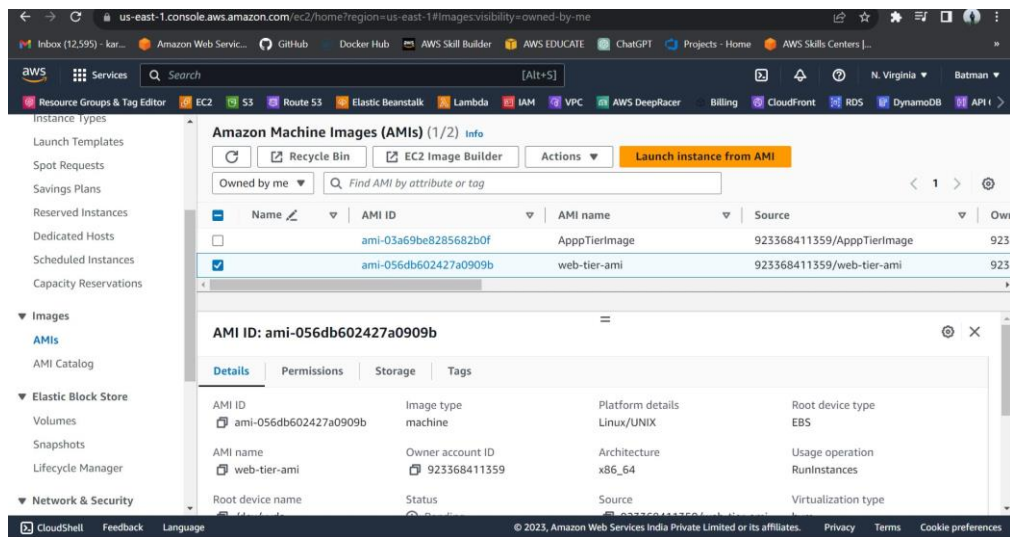
Now when I plug in the public IP of our web tier instance, I should see my website and in that I can also see the database working.

Step.9: External Load Balancer and Auto Scaling:

I created an Amazon Machine Image (AMI) of the web tier instance and used that to set up autoscaling with an external facing load balancer in order to make this tier highly available.

Web Tier AMI:

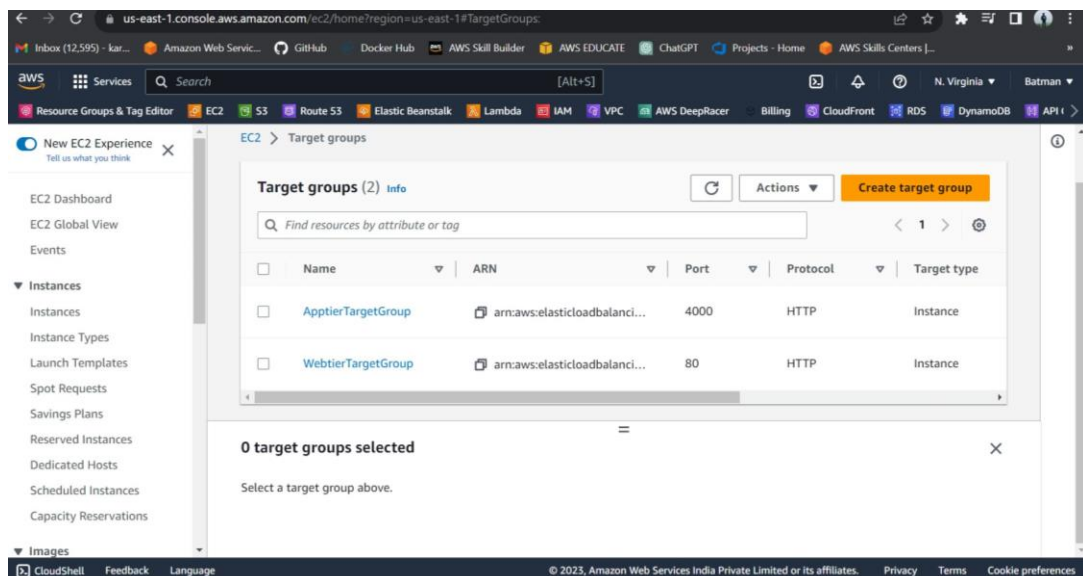
I created an Image of the web-tier instance.



Target Group:

I created our target group to use with the load balancer. The purpose of forming this target group is to use with our load balancer so it may balance traffic across our public web tier instances.

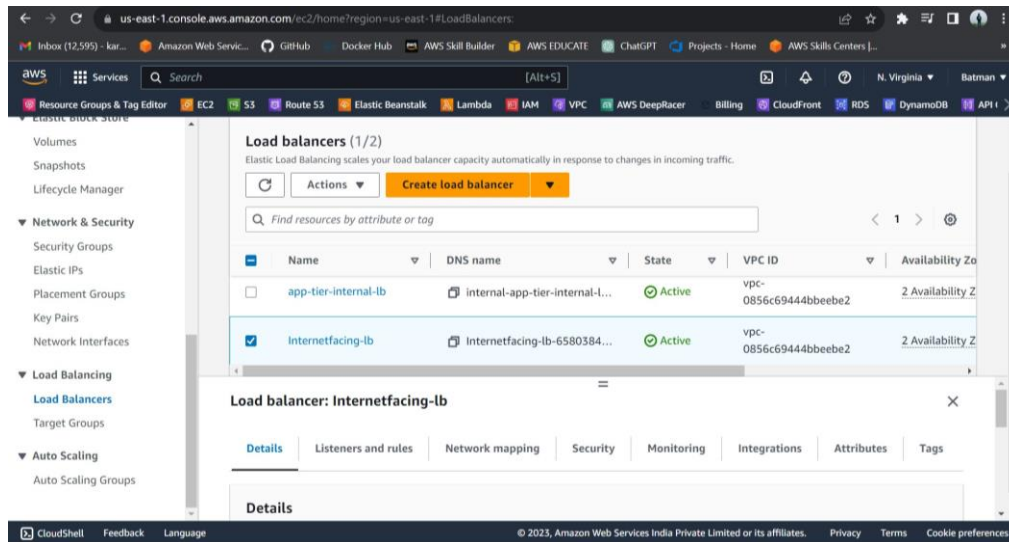
Then, I set the protocol to **HTTP** and the port to 80, this is the port NGINX is listening on, then changed the health check path to be **/health**. I did not register any targets for now.



Internet Facing Load Balancer: I created an Application Load Balancer for our **HTTP** traffic and selected **internet facing** since this one will not be public facing, but

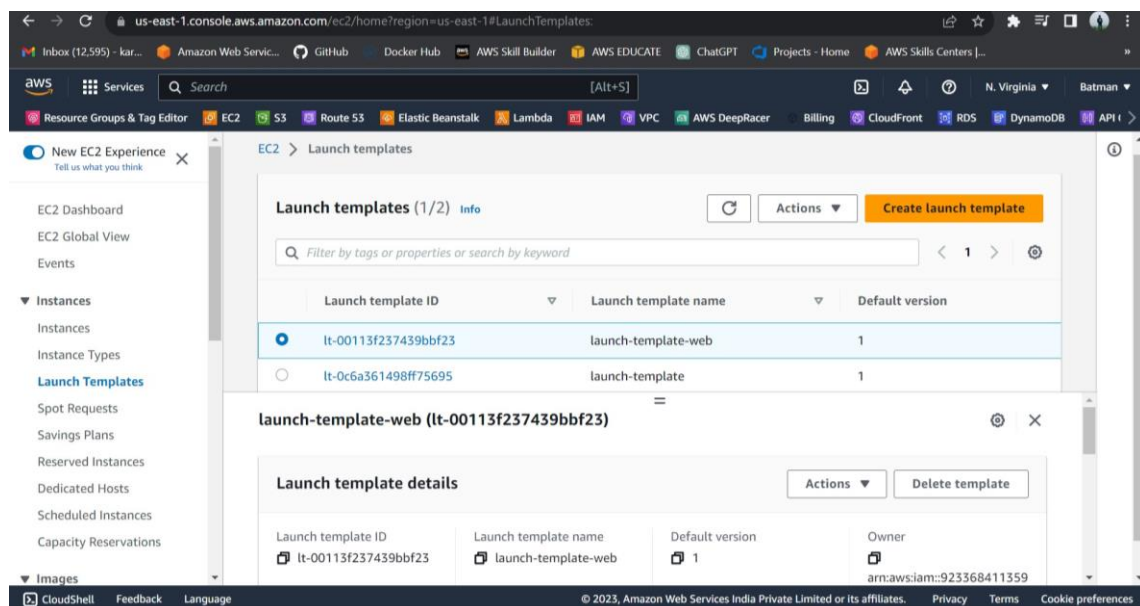
rather it will route traffic from our web tier to the app tier. Then selected the VPC and public subnets.

Then selected the security group I created for this internal ALB. Now, this ALB will be listening for HTTP traffic on port 80. It will be forwarding the traffic to our target group that I just created.



Launch Template:

I created a Launch template with the AMI I created earlier with the app tier. For Key pair and Network Settings, I didn't include it in the template. Since we don't need a key pair to access our instances and we'll be setting the network information in the autoscaling group. Then I used the security group for our web tier, and the same IAM role I have been using for our EC2 instances.



Auto Scaling:

I now created the Auto Scaling Group for our web instances by setting my VPC, and the public subnets for the web tier. Then attached this Auto Scaling Group to the Load Balancer I just created by selecting the existing web tier load balancer's target group. I set desired, minimum and maximum capacity to 2.

I now have our external load balancer and autoscaling group configured correctly.

I should see the autoscaling group spinning up 2 new web tier instances. To test if this is working correctly, I deleted one of our new instances manually and waited to see if a new instance is booted up to replace it.

To test if my entire architecture is working, I navigated to our external facing load balancer, and plugged in the DNS name into my browser.

