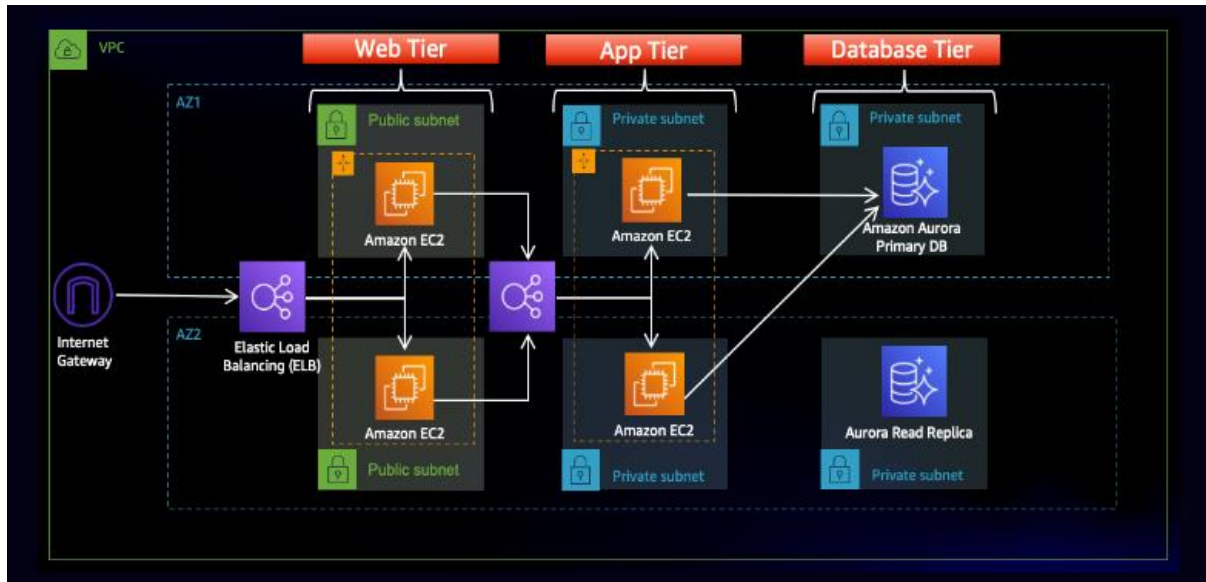


Three Tier Web Architecture:

Introduction:

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 webservers 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.

Part 0: Setup

For this project, we will be downloading the code from Github and upload it to S3 so our instances can access it. We will also create an AWS Identity and Access Management EC2 role so we can use AWS Systems Manager Session Manager to connect to our instances securely and without needing to create SSH key pairs.

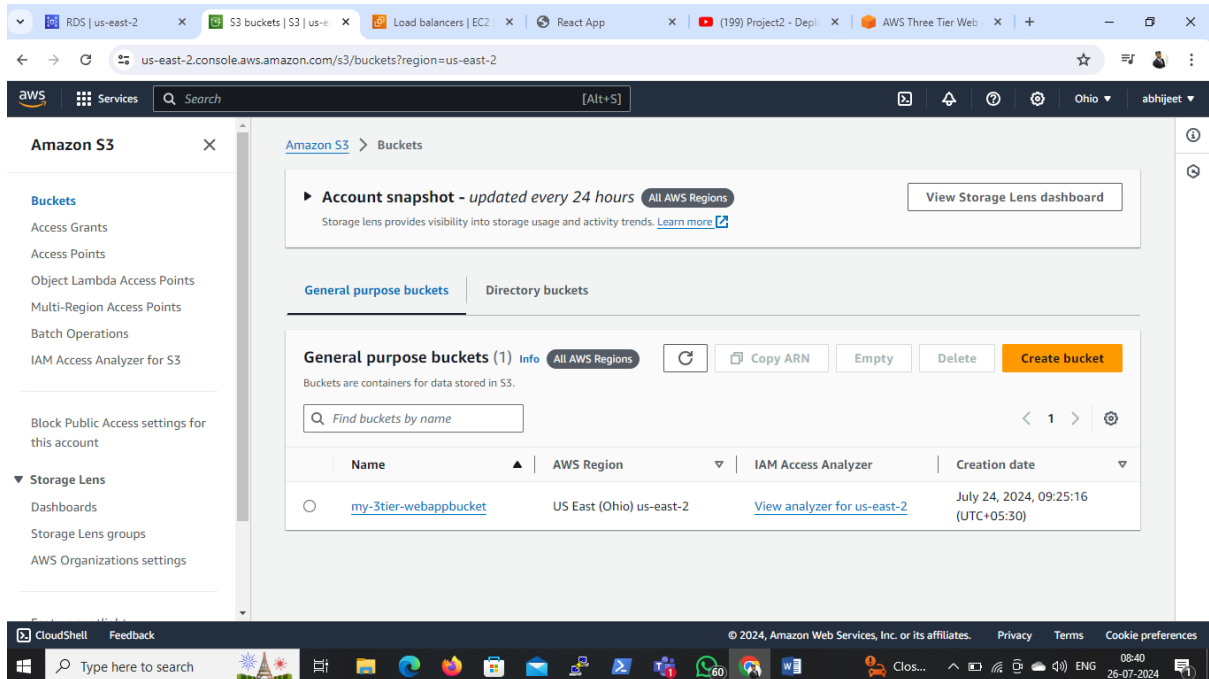
Download Code from Github

Download the code from [this repository](https://github.com/aws-samples/aws-three-tier-web-architecture-workshop) into your local environment by running the command below. If you don't have git installed, you can just download the zip. Save it somewhere you can easily access.

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

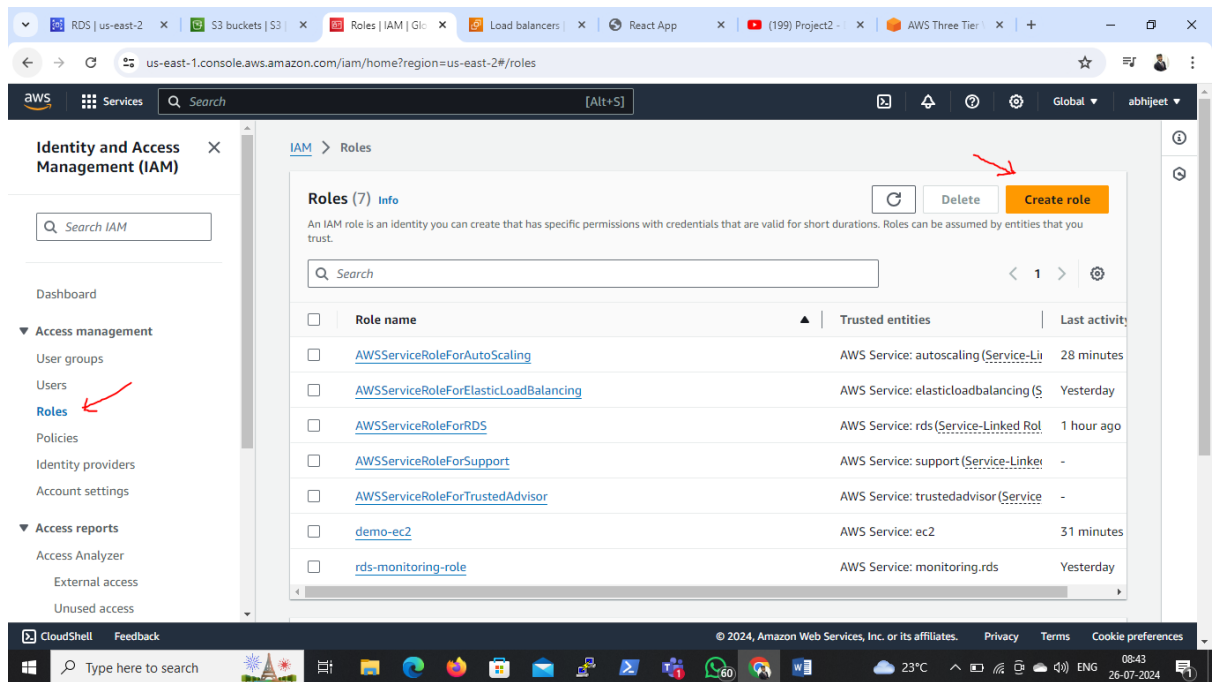
S3 Bucket Creation

1. Navigate to the S3 service in the AWS console and create a new S3 bucket.
2. Give it a unique name, and then leave all the defaults as in. Make sure to select the region that you intend to run this whole lab in. This bucket is where we will upload our code later.

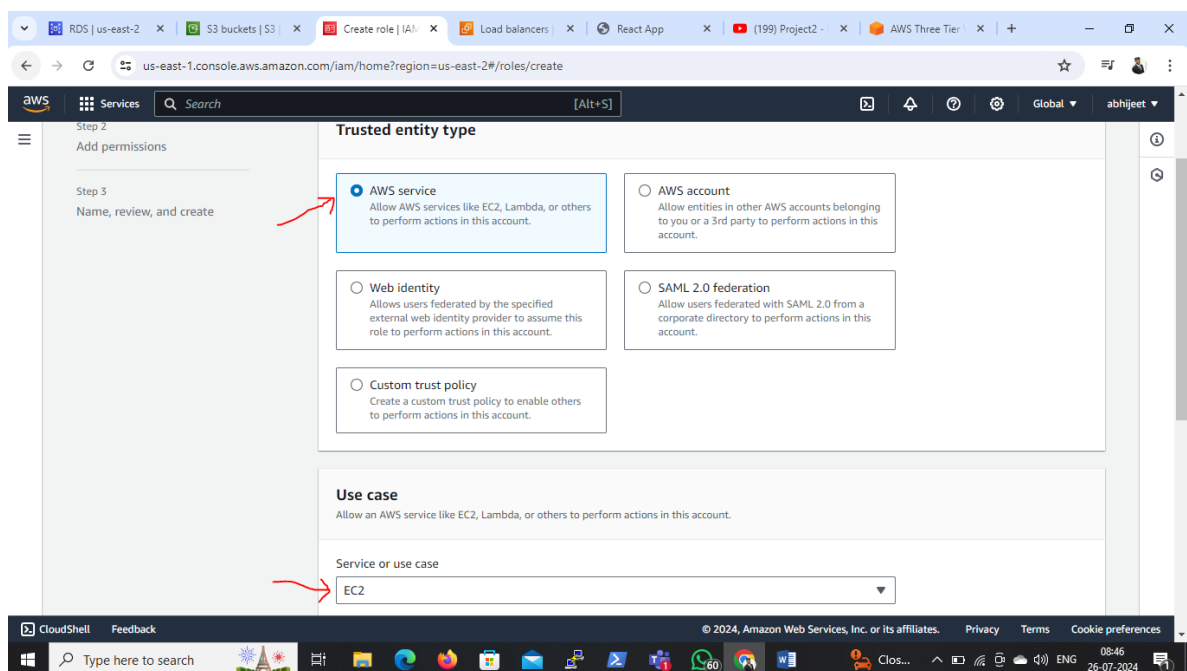


IAM EC2 Instance Role Creation

1. Navigate to the IAM dashboard in the AWS console and create an EC2 role.

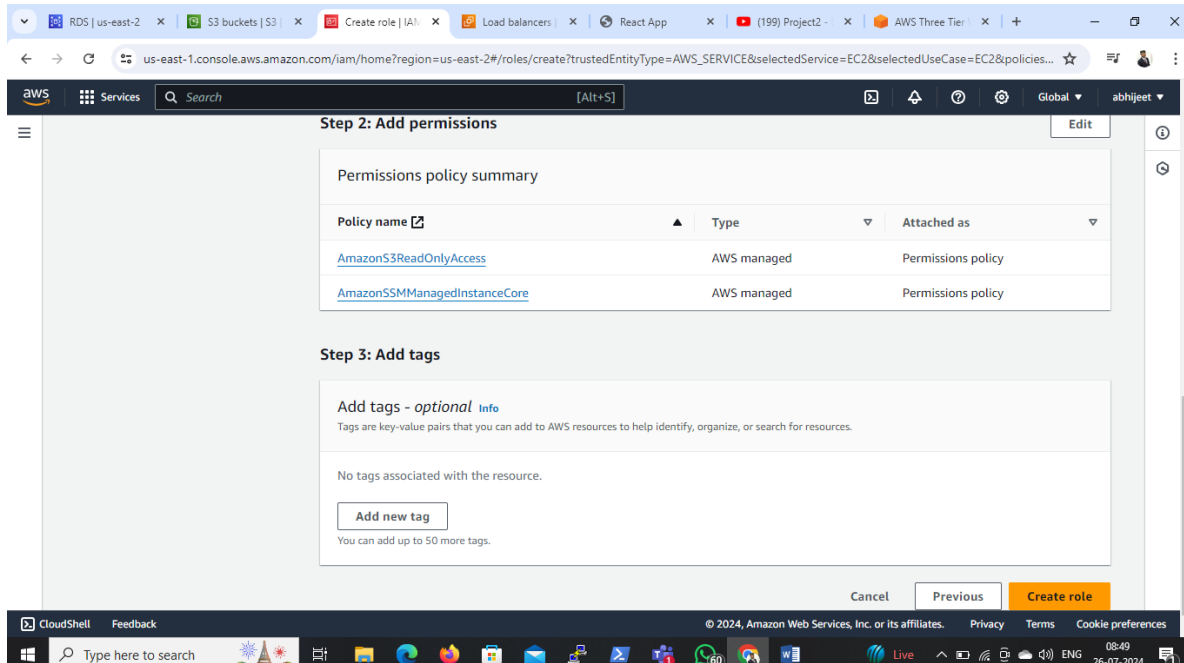


2. Select EC2 as the trusted entity.



3. When adding permissions, include the following AWS managed policies. We can search for them and select them. 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.

- **AmazonSSMManagedInstanceCore**
- **AmazonS3ReadOnlyAccess**



4. Give your role a name, and then click **Create Role**.

Part 1: Networking and Security

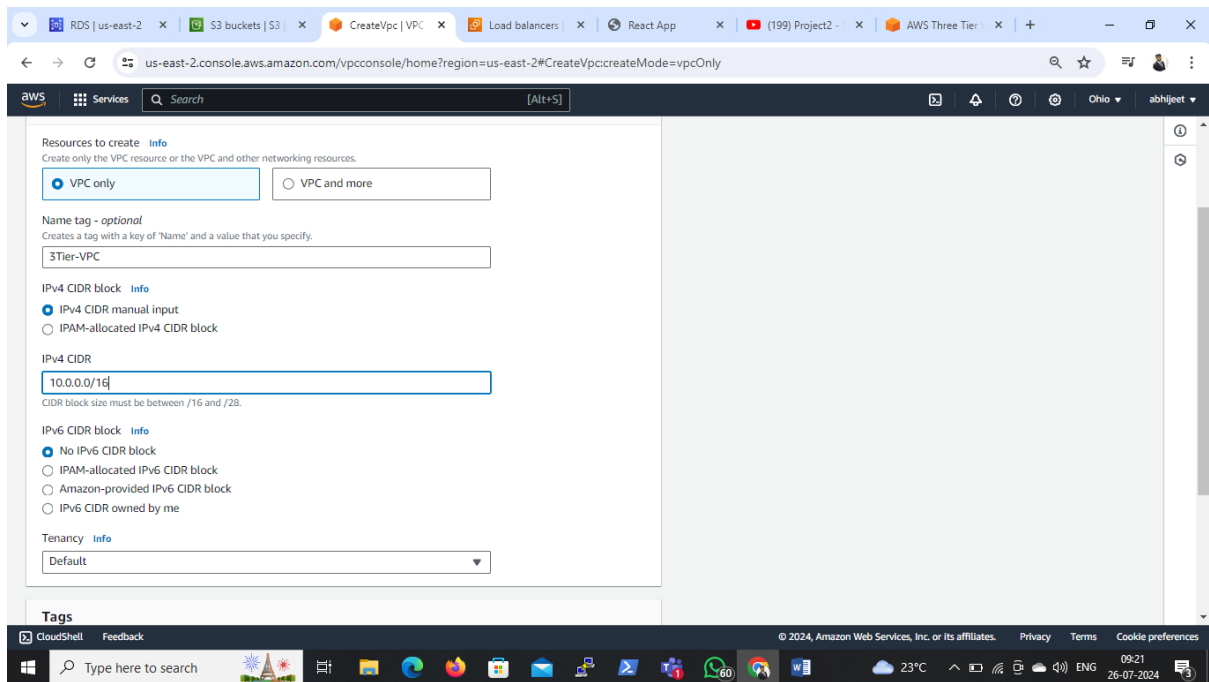
In this section we will be building out the VPC networking components as well as security groups that will add a layer of protection around our EC2 instances, Aurora databases, and Elastic Load Balancers.

VPC Creation

1. Navigate to the VPC dashboard in the AWS console and navigate to **Your VPCs** on the left hand side.
2. Make sure **VPC only** is selected, and fill out the VPC Settings with a **Name tag** and a **CIDR range** of your choice.

***NOTE:** Make sure you pay attention to the region you're deploying all your resources in. You'll want to stay consistent for this workshop.*

***NOTE:** Choose a CIDR range that will allow you to create at least 6 subnets.*



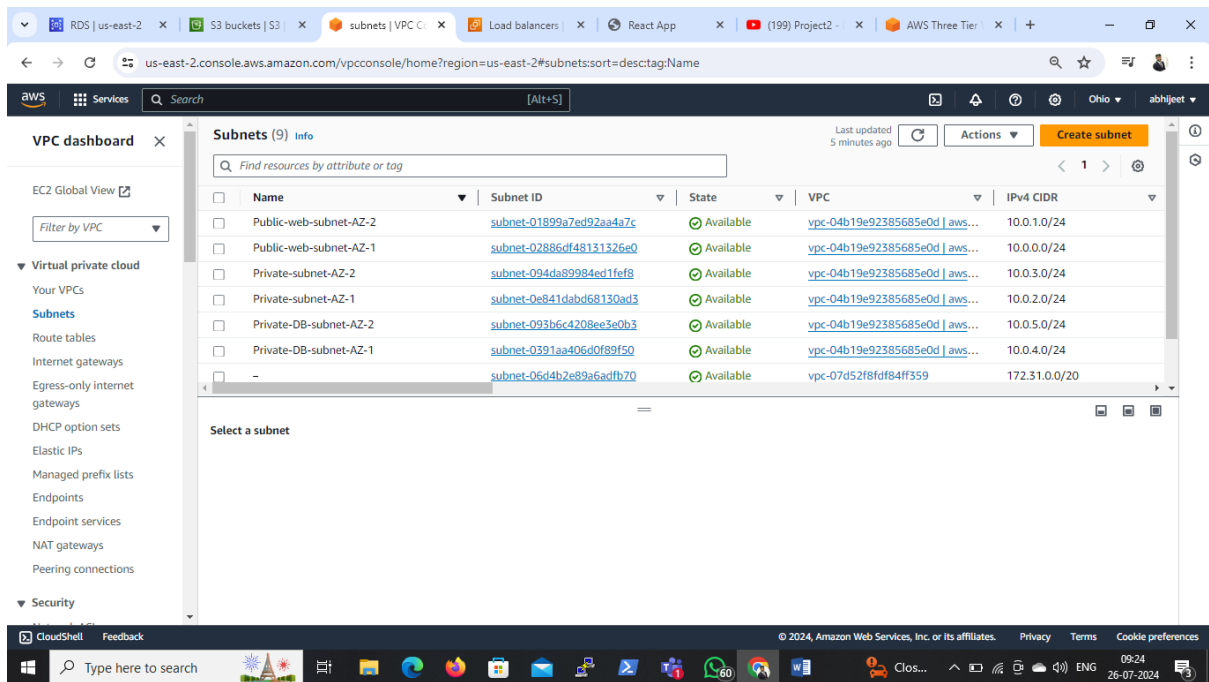
Subnet Creation

1. Next, create your subnets by navigating to **Subnets** on the left side of the dashboard and clicking **Create subnet**.
2. We will need **six** subnets across **two** availability zones. That means that **three** subnets will be in one availability zone, and three subnets will be in another zone. Each subnet in one availability zone will correspond to one layer of our three tier architecture. Create each of the 6 subnets by specifying the VPC we created in part 1 and then choose a name, availability zone, and appropriate CIDR range for each of the subnets.

*NOTE: It may be helpful to have a naming convention that will help we remember what each subnet is for. For example in one AZ you might have the following: **Public-Web-Subnet-AZ-1**, **Private-App-Subnet-AZ-1**, **Private-DB-Subnet-AZ-1**.*

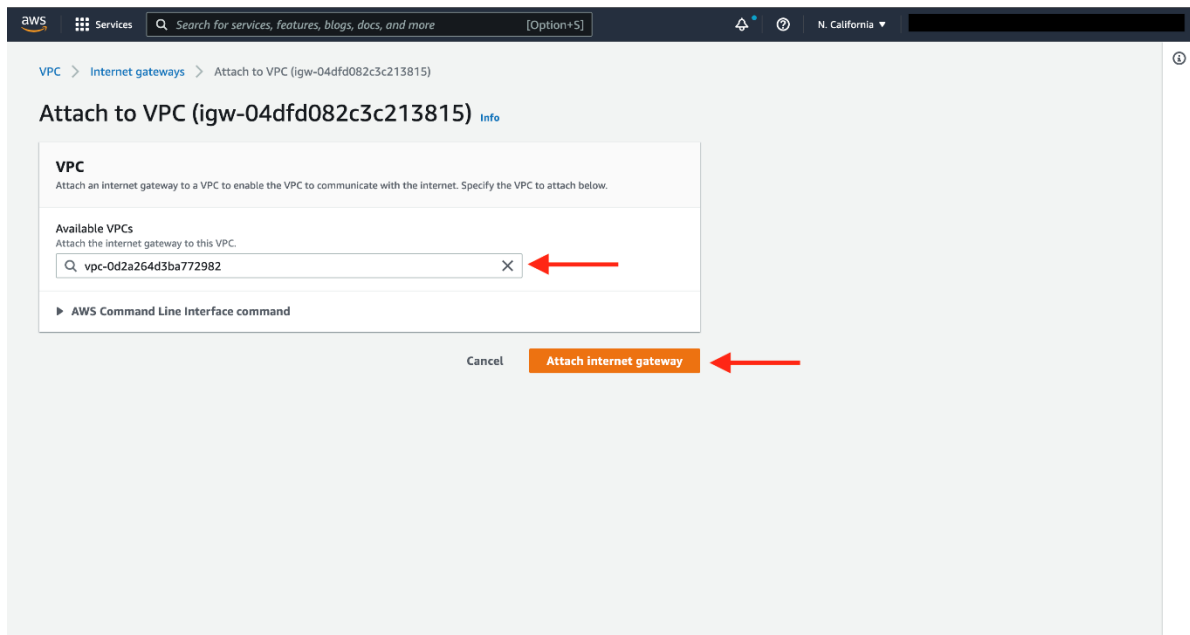
NOTE: Remember, CIDR range for the subnets will be subsets of your VPC CIDR range.

Our final subnet setup should be similar to this. Verify that you have 3 subnets across 2 different availability zones.



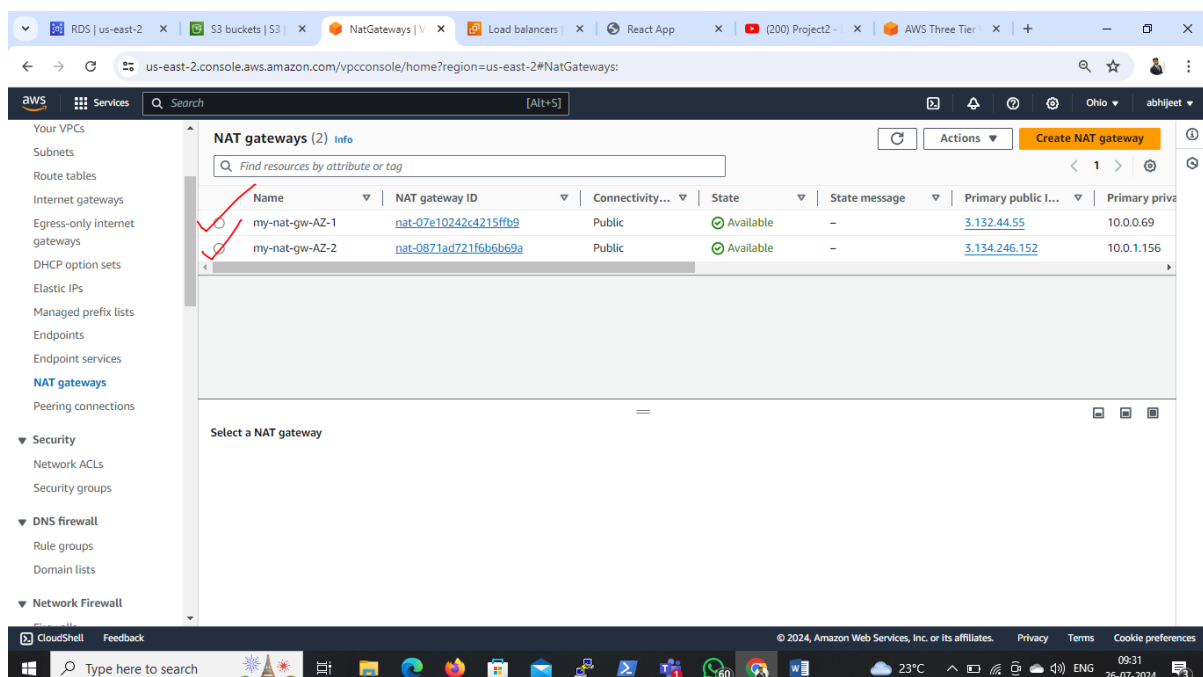
Internet Gateway

1. In order to give the public subnets in our VPC internet access we will have to create and attach an Internet Gateway. On the left hand side of the VPC dashboard, select **Internet Gateway**.
2. Create internet gateway by simply giving it a name and clicking **Create internet gateway**.
3. After creating the internet gateway, attach it to our VPC that we create in the **VPC and Subnet Creation** step of the workshop. We have a couple options on how to do this, either with the creation success message or the **Actions** drop down.
4. Then, select the correct VPC and click **Attach internet gateway**.



NAT Gateway

1. 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, you'll deploy one NAT gateway in each of your **public** subnets. Navigate to **NAT Gateways** on the left side of the current dashboard and click **Create NAT Gateway**.
2. Fill in the **Name**, choose one of the **public subnets** you created in part 2, and then allocate an Elastic IP. Click **Create NAT gateway**.
3. Repeat step 1 and 2 for the other subnet.



Routing Configuration

1. Navigate to **Route Tables** on the left side of the VPC dashboard and click **Create route table**. First, let's create one route table for the web layer *public subnets* and name it accordingly.
2. After creating the route table, you'll automatically be taken to the details page. Scroll down and click on the **Routes tab** and **Edit routes**.
3. Add a route that directs traffic from the VPC to the internet gateway. In other words, for all traffic *destined* for IPs outside the VPC CIDR range, add an entry that directs it to the internet gateway as a *target*. Save the changes.
4. Edit the *Explicit Subnet Associations* of the route table by navigating to the route table details again. Select **Subnet Associations** and click **Edit subnet associations**.
5. Select the two web layer public subnets we created earlier and click **Save associations**.
6. Now create 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, so add the appropriate routes for that.
7. Once the route tables are created and routes added, add the appropriate subnet associations for each of the app layer private subnets.

Security Groups

1. Security groups will tighten the rules around which traffic will be allowed to our Elastic Load Balancers and EC2 instances. Navigate to **Security Groups** on the left side of the VPC dashboard, under **Security**.

The screenshot displays the AWS Management Console interface for the 'Security Groups' page. The left-hand navigation pane shows the 'SECURITY' section expanded, with 'Security Groups' selected and highlighted by a red arrow. The main content area, titled 'Security Groups (1) Info', shows a table of security groups associated with the VPC ID 'vpc-0d2a264d3ba772982'. The table has columns for Name, Security group ID, Security group name, VPC ID, Description, and Owner. One security group is listed with the name 'default' and ID 'sg-0bf79a59048818994'. In the top right corner of the main content area, there is a red arrow pointing to an orange button labeled 'Create security group'.

Name	Security group ID	Security group name	VPC ID	Description	Owner
-	sg-0bf79a59048818994	default	vpc-0d2a264d3ba772982	default VPC security gr...	192462669998

- The first security group you'll create is for the public, **internet facing** load balancer. After typing a name and description, add an inbound rule to allow **HTTP** type traffic for your **IP**.

The screenshot shows the 'Create security group' page in the AWS Management Console. The 'Basic details' section has three fields: 'Security group name' with the value 'internet-facing-lb-sg', 'Description' with 'External load balancer security group', and 'VPC' with 'vpc-0d2a264d3ba772982'. Red arrows point to each of these fields. The 'Inbound rules' section shows a table with one rule. The 'Type' column has a dropdown menu with 'HTTP' selected, indicated by a red arrow. The 'Protocol' column has 'TCP', 'Port range' has '80', and 'Source' has 'My IP', also indicated by a red arrow. A search box next to 'My IP' shows a redacted IP address. There is a 'Delete' button and an 'Add rule' button at the bottom of the table.

Type	Protocol	Port range	Source	Description - optional
HTTP	TCP	80	My IP	

- The second security group you'll create is for the public instances in the web tier. After typing a name and description, add an inbound rule that allows **HTTP** type traffic from your internet facing load balancer security group you created in the previous step. This will allow traffic from your public facing load balancer to hit your instances. Then, add an additional rule that will allow HTTP type traffic for your IP. This will allow you to access your instance when we test.

The screenshot shows the 'Create security group' page in the AWS Management Console. The 'Basic details' section has three fields: 'Security group name' with the value 'WebTierSG', 'Description' with 'SG for the Web Tier', and 'VPC' with 'vpc-0d2a264d3ba772982'. Red arrows point to each of these fields. The 'Inbound rules' section shows a table with two rules. The first rule has 'Type' as 'HTTP', 'Protocol' as 'TCP', 'Port range' as '80', and 'Source' as 'My IP'. The second rule has 'Type' as 'HTTP', 'Protocol' as 'TCP', 'Port range' as '80', and 'Source' as 'Custom'. Red arrows point to the 'Type' dropdown of both rules. A search box next to 'Custom' shows a dropdown menu with the value 'sg-072fe91349cba745f' selected, indicated by a red arrow.

Type	Protocol	Port range	Source	Description - optional
HTTP	TCP	80	My IP	
HTTP	TCP	80	Custom	

4. The third security group will be for our internal load balancer. Create this new security group and add an inbound rule that allows **HTTP** type traffic from your public instance security group. This will allow traffic from your web tier instances to hit your internal load balancer.

The screenshot shows the 'Create security group' page in the AWS Management Console. The 'Basic details' section has the following fields filled out: 'Security group name' is 'internal-lb-sg', 'Description' is 'SG for the internal load balancer', and 'VPC' is 'vpc-0d2a264d3ba772982'. The 'Inbound rules' section has a table with one rule: Type 'HTTP', Protocol 'TCP', Port range '80', and Source 'Custom'. A dropdown menu is open for the 'Source' field, showing a list of security groups. The 'WebTierSG | sg-0d115f758786138a5' is highlighted with a red box. A red arrow points to the 'Add rule' button.

5. The fourth security group we'll configure is for our private instances. After typing a name and description, add an inbound rule that will allow **TCP** type traffic on port **4000** from the **internal load balancer security group** you created in the previous step. 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. You should also add another route for port **4000** that allows **your IP** for testing.

The screenshot shows the 'Create security group' page in the AWS Management Console. The 'Basic details' section has the following fields filled out: 'Security group name' is 'PrivateInstanceSG', 'Description' is 'SG for our private app tier sg', and 'VPC' is 'vpc-0d2a264d3ba772982'. The 'Inbound rules' section has a table with two rules: Type 'Custom TCP', Protocol 'TCP', Port range '4000', and Source 'Custom'; and Type 'Custom TCP', Protocol 'TCP', Port range '4000', and Source 'My IP'. A dropdown menu is open for the 'Source' field of the first rule, showing a list of security groups. The 'internal-lb-sg | sg-022b4036ffff83a40' is highlighted with a red box. A red arrow points to the 'Add rule' button.

6. The fifth security group we'll configure protects our private database instances. For this security group, add an inbound rule that will allow traffic from the private instance security group to the MYSQL/Aurora port (3306).

Create security group [Info](#)

A security group acts as a virtual firewall for your instance to control inbound and outbound traffic. To create a security group, you must specify a VPC. The security group must be in the same VPC as the instance it protects. You can create a security group before or after you create an instance. If you create a security group before you create an instance, you can associate the security group with the instance when you create it. If you create a security group after you create an instance, you can associate the security group with the instance later.

Basic details

Security group name [Info](#)
DBSG
Name cannot be edited after creation.

Description [Info](#)
SG for our databases

VPC [Info](#)
vpc-0d2a264d3ba772982

Inbound rules [Info](#)

Type Info	Protocol Info	Port range Info	Source Info	Description - optional Info
MYSQL/Aurora	TCP	3306	Custom	

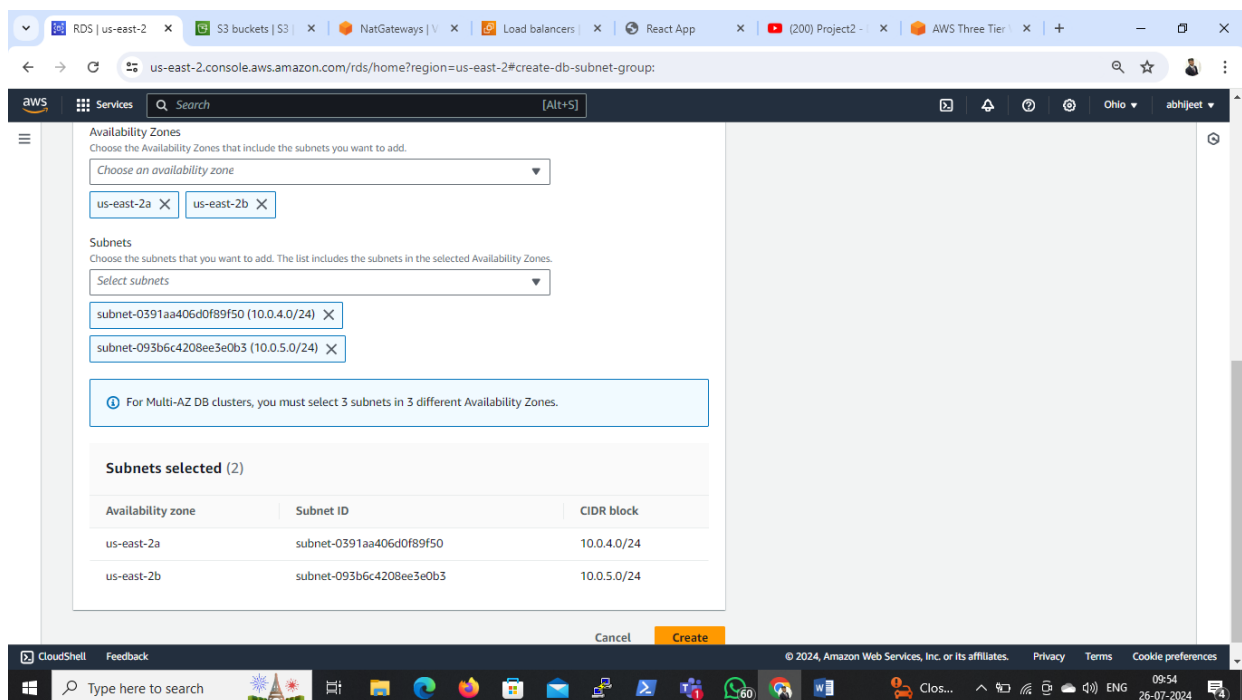
Outbound rules [Info](#)

Part 2: Database Deployment

This section of the project will walk through deploying the database layer of the three tier architecture.

Subnet Groups

1. Navigate to the RDS dashboard in the AWS console and click on **Subnet groups** on the left hand side. Click **Create DB subnet group**.
2. Give subnet group a name, description, and choose the VPC we created.
3. When adding subnets, make sure to add the subnets we created in each availability zone specifically for our database layer. You may have to navigate back to the VPC dashboard and check to make sure you're selecting the correct subnet IDs.



Database Deployment

1. Navigate to **Databases** on the left hand side of the RDS dashboard and click **Create database**.
2. We'll now go through several configuration steps. Start with a **Standard create** for this **MySQL-Compatible Amazon Aurora** database. Leave the rest of the defaults in the **Engine options** as default.
3. Under the **Templates** section choose **Dev/Test** since this isn't being used for production at the moment. Under **Settings** set a username and password of your

choice and note them down since we'll be using password authentication to access our database.

4. Next, under **Availability and durability** change the option to create an Aurora Replica or reader node in a different availability zone. Under **Connectivity**, set the VPC, choose the subnet group we created earlier, and select no for public access.
5. Set the security group we created for the database layer, make sure **password authentication** is selected as our authentication choice, and create the database.
6. When your database is provisioned, you should see a reader and writer instance in the database subnets of each availability zone. Note down the writer endpoint for your database for later use.

The screenshot shows the AWS Management Console for Amazon RDS. The left sidebar contains navigation links for Dashboard, Databases, Query Editor, Performance Insights, Snapshots, Exports in Amazon S3, Automated backups, Reserved instances, Proxies, Subnet groups, Parameter groups, Option groups, Custom engine versions, Zero-ETL integrations, Events, and Event subscriptions. The main content area shows the 'Databases (3)' page with a 'Create database' button. A table lists the databases:

DB identifier	Status	Role	Engine	Region & AZ	Size
database-1	Stopped temporarily	Regional cluster	Aurora MySQL	us-east-2	2 instances
database-1-instance-1	Stopped temporarily	Writer instance	Aurora MySQL	us-east-2b	db.r6g.2xlarge
database-1-instance-1-us-east-2a	Stopped temporarily	Reader instance	Aurora MySQL	us-east-2a	db.r6g.2xlarge

Part 3: App Instance Deployment

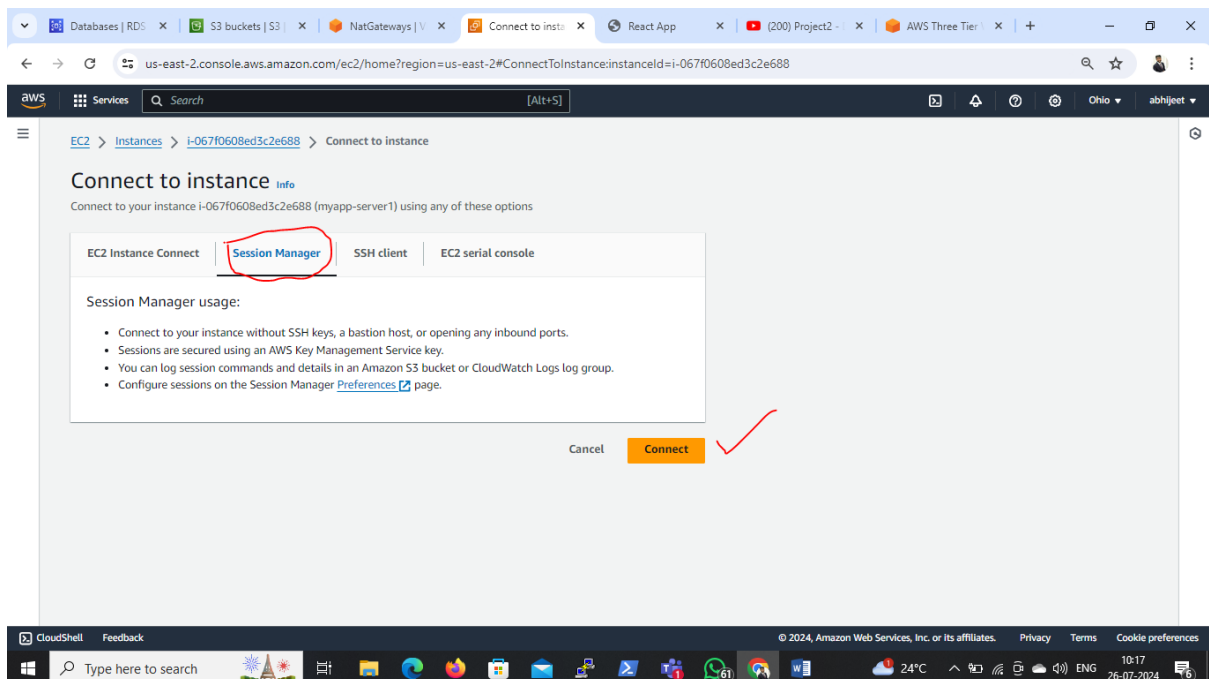
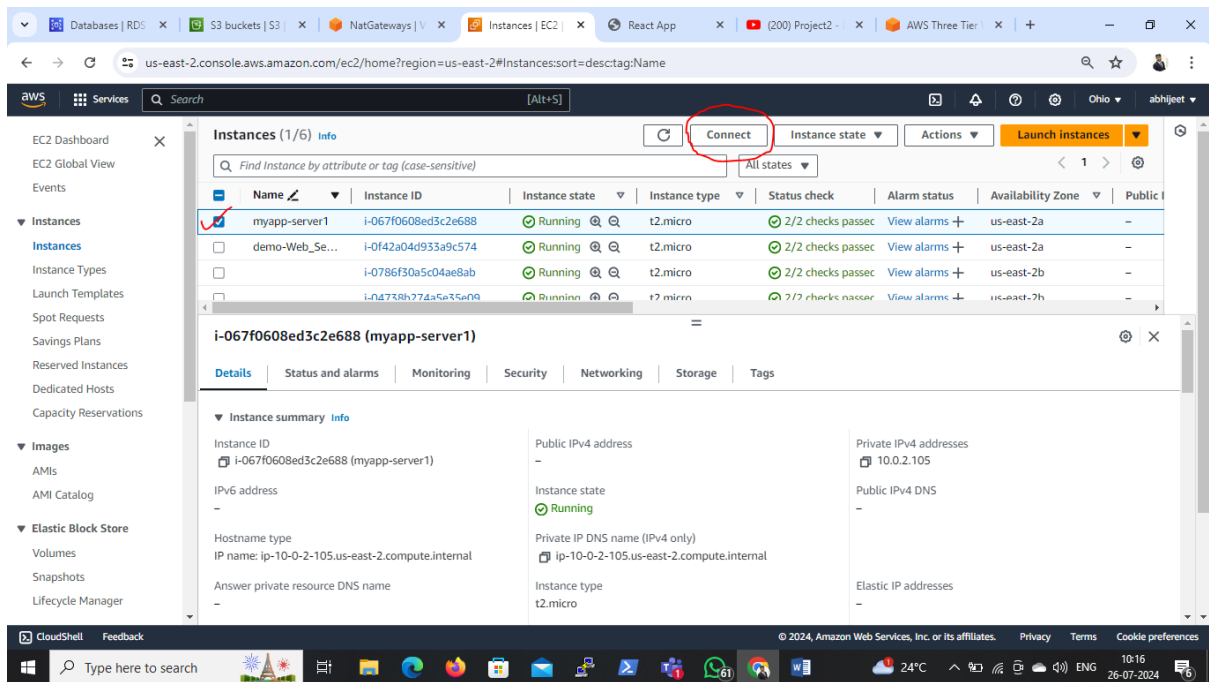
In this section of our project we will create an EC2 instance for our app layer and make all necessary software configurations so that the app can run. The app layer consists of a Node.js application that will run on port 4000. We will also configure our database with some data and tables.

App Instance Deployment

1. Navigate to the EC2 service dashboard and click on **Instances** on the left hand side. Then, click **Launch Instances**.
2. Select the first **Amazon Linux 2 AMI**
3. We'll be using the free tier eligible **T.2 micro** instance type. Select that and click **Next: Configure Instance Details**.
4. When configuring the instance details, make sure to select to correct **Network**, **subnet**, and **IAM role** we created. Note that this is the app layer, so use one of the private subnets we created for this layer.
5. We'll be keeping the defaults for storage so click next twice. When you get to the tag screen input a **Name** as a key and call the instance AppLayer. It's a good idea to tag your instances so you can easily keep track of what each instance was created for. Click **Next: Configure Security Group**.
6. Earlier we created a security group for our private app layer instances, so go ahead and select that in this next section. Then click **Review and Launch**. Ignore the warning about connecting to port 22- we don't need to do that.
7. When you get to the **Review Instance Launch** page, review the details you configured and click **Launch**. You'll see a pop up about creating a key pair. Since we are using Systems Manager Session Manager to connect to the instance, **proceed without a keypair**. Click **Launch**.

Connect to Instance

1. Navigate to list of running EC2 Instances by clicking on **Instances** on the left hand side of the EC2 dashboard. When the instance state is running, connect to your instance by clicking the checkmark box to the left of the instance, and click the connect button on the top right corner of the dashboard. Select the Session Manager tab, and click connect. This will open a new browser tab for you.

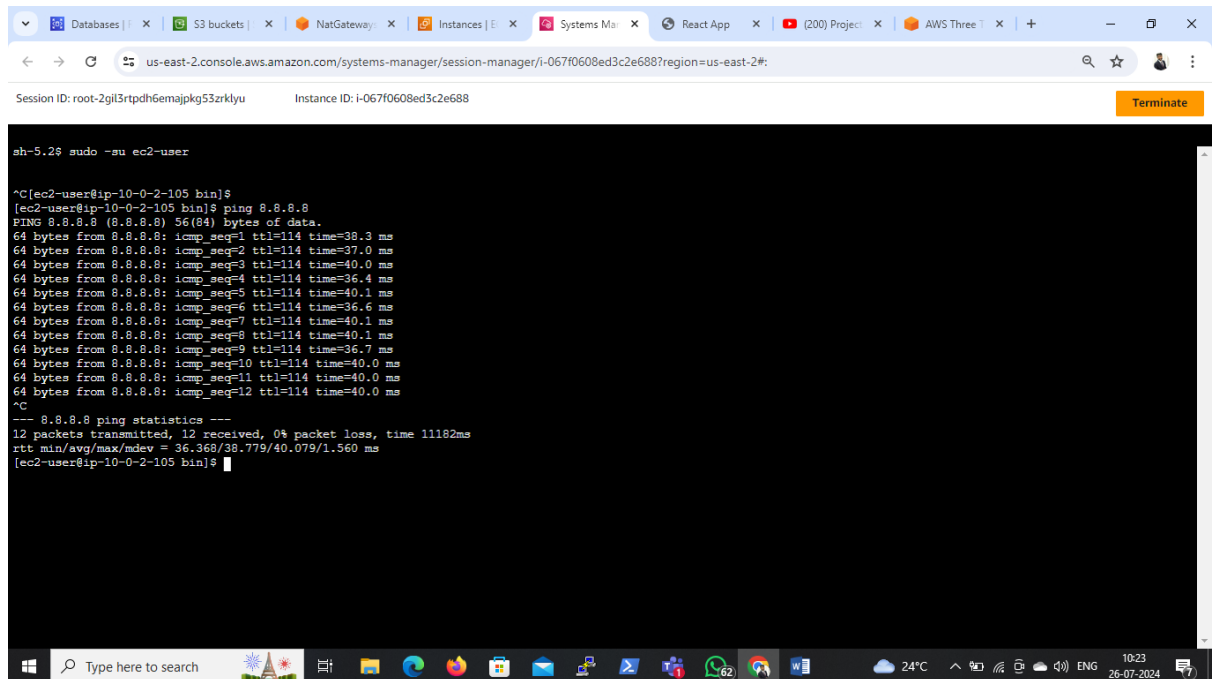


- When you first connect to your instance like this, you will be logged in as ssm-user which is the default user. Switch to ec2-user by executing the following command in the browser terminal:

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

- Let's take this moment to make sure that we are able to reach the internet via our NAT gateways. If your network is configured correctly up till this point, you should be able to ping the google DNS servers:

ping 8.8.8.8



The screenshot shows a web browser window displaying an AWS Systems Manager session. The browser tabs include 'Databases', 'S3 buckets', 'NatGateway', 'Instances', 'Systems Ma...', 'React App', '(200) Proj...', and 'AWS Three'. The address bar shows the URL 'us-east-2.console.aws.amazon.com/systems-manager/session-manager/i-067f0608ed3c2e688?region=us-east-2#'. Below the browser window, a terminal window is open, showing the following commands and output:

```
sh-5.2$ sudo -su ec2-user
^C[ec2-user@ip-10-0-2-105 bin]$
[ec2-user@ip-10-0-2-105 bin]$ ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data:
64 bytes from 8.8.8.8: icmp_seq=1 ttl=114 time=38.3 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=114 time=37.0 ms
64 bytes from 8.8.8.8: icmp_seq=3 ttl=114 time=40.0 ms
64 bytes from 8.8.8.8: icmp_seq=4 ttl=114 time=36.4 ms
64 bytes from 8.8.8.8: icmp_seq=5 ttl=114 time=40.1 ms
64 bytes from 8.8.8.8: icmp_seq=6 ttl=114 time=36.6 ms
64 bytes from 8.8.8.8: icmp_seq=7 ttl=114 time=40.1 ms
64 bytes from 8.8.8.8: icmp_seq=8 ttl=114 time=40.1 ms
64 bytes from 8.8.8.8: icmp_seq=9 ttl=114 time=36.7 ms
64 bytes from 8.8.8.8: icmp_seq=10 ttl=114 time=40.0 ms
64 bytes from 8.8.8.8: icmp_seq=11 ttl=114 time=40.0 ms
64 bytes from 8.8.8.8: icmp_seq=12 ttl=114 time=40.0 ms
^C
-- 8.8.8.8 ping statistics --
12 packets transmitted, 12 received, 0% packet loss, time 11182ms
rtt min/avg/max/mdev = 36.368/38.779/40.079/1.560 ms
[ec2-user@ip-10-0-2-105 bin]$
```

We should see a transmission of packets. Stop it by pressing **ctrl c**.

Configure Database

1. Start by downloading the MySQL CLI:

```
sudo yum install mysql -y
```

2. Initiate our DB connection with your Aurora RDS writer endpoint. In the following command, replace the RDS writer endpoint and the username, and then execute it in the browser terminal:

```
mysql -h CHANGE-TO-OUR-RDS-ENDPOINT -u CHANGE-TO-USER-NAME  
-p
```

We will then be prompted to type in your password. Once you input the password and hit enter, you should now be connected to your database.

3. Create a database called **webappdb** with the following command using the MySQL CLI:

```
CREATE DATABASE webappdb;
```

We can verify that it was created correctly with the following command:

SHOW DATABASES;

4. Create a data table by first navigating to the database we just created:

USE webappdb;

Then, create 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));**

Verify the table was created:

SHOW TABLES;

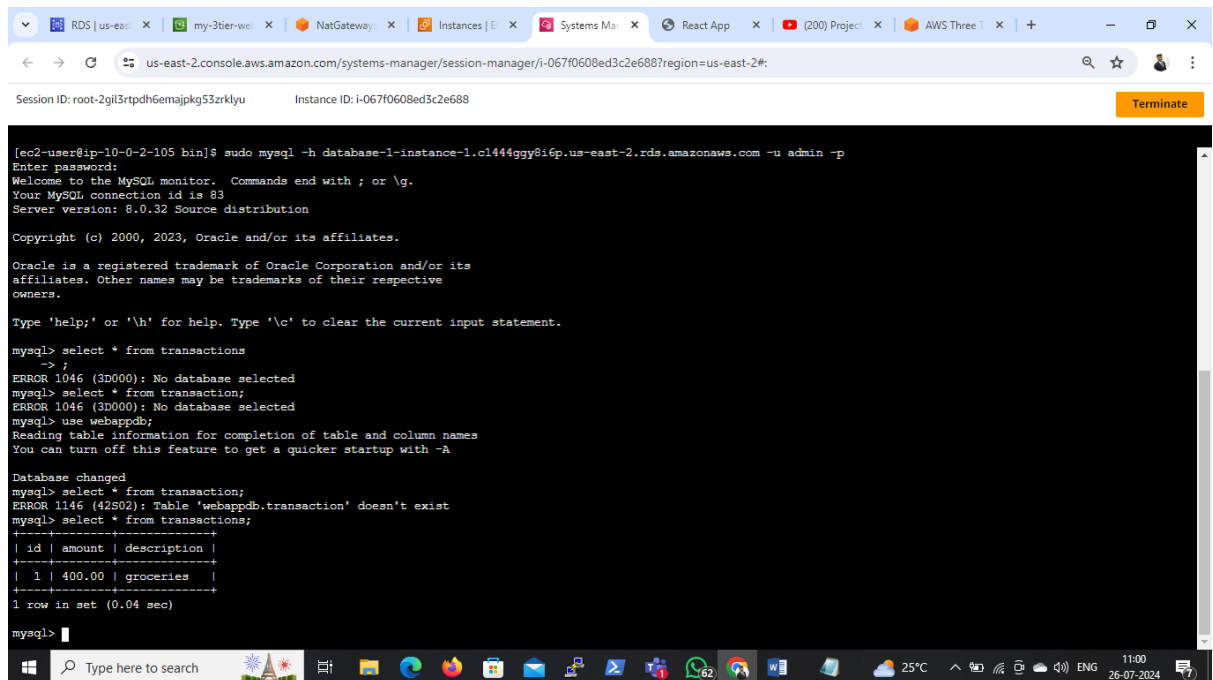
5. Insert data into table for use/testing later:

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

6. Verify that your data was added by executing the following command:

SELECT * FROM transactions;

7. When finished, just type **exit** and hit enter to exit the MySQL client.



```
[ec2-user@ip-10-0-2-105 bin]$ sudo mysql -h database-1-instance-1.ci44ggy816p.us-east-2.rds.amazonaws.com -u admin -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 83
Server version: 8.0.32 Source distribution

Copyright (c) 2000, 2023, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> select * from transactions
-> ;
ERROR 1046 (3D000): No database selected
mysql> select * from transaction;
ERROR 1046 (3D000): No database selected
mysql> use webappdb;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> select * from transaction;
ERROR 1146 (42S02): Table 'webappdb.transaction' doesn't exist
mysql> select * from transactions;
+----+-----+-----+
| id | amount | description |
+----+-----+-----+
| 1  | 400.00 | groceries   |
+----+-----+-----+
1 row in set (0.04 sec)

mysql>
```

Configure App Instance

1. The first thing we will do is update our database credentials for the app tier. To do this, open the **application-code/app-tier/DbConfig.js** file from the github repo in your favorite text editor on your computer. You'll see empty strings for the hostname, user, password and database. Fill this in with the credentials you configured for your database, the **writer** endpoint of your database as the hostname, and **webappdb** for the database. Save the file.
2. Upload the **app-tier** folder to the S3 bucket that you created in part 0.
3. Go back to your SSM session. Now we need to install all of the necessary components to run our backend application. Start by installing NVM (node version manager).

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

```
source ~/.bashrc
```

4. Next, install a compatible version of Node.js and make sure it's being used

```
nvm install 16
```

```
nvm use 16
```

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

```
npm install -g pm2
```

6. Now we need to download our code from our s3 buckets onto our instance. In the command below, replace BUCKET_NAME with the name of the bucket you uploaded the **app-tier** folder to:

```
cd ~/
```

```
aws s3 cp s3://BUCKET_NAME/app-tier/ app-tier --recursive
```

7. Navigate to the app directory, install dependencies, and start the app with pm2.

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

```
npm install
```

```
pm2 start index.js
```

To make sure the app is running correctly run the following:

```
pm2 list
```

If you see a status of online, the app is running. If you see errored, then you need to do some troubleshooting. To look at the latest errors, use this command:

```
pm2 logs
```

8. Right now, pm2 is just making sure our app stays running when we 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:

```
pm2 startup
```

After running this you will see a message similar to this.

[PM2] To setup the Startup Script, copy/paste the following command: `sudo env PATH=$PATH:/home/ec2-user/.nvm/versions/node/v16.0.0/bin /home/ec2-user/.nvm/versions/node/v16.0.0/lib/node_modules/pm2/bin/pm2 startup systemd -u ec2-user —hp /home/ec2-user`

DO NOT run the above command, rather you should copy and past the command in the output you see in your own terminal. After you run it, save the current list of node processes with the following command:

```
pm2 save
```

Test App Tier

Now let's run a couple tests to see if our app is configured correctly and can retrieve data from the database.

To hit out health check endpoint, copy this command into your SSM terminal. This is our simple health check endpoint that tells us if the app is simply running.

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

Next, test your database connection. You can do that by hitting the following endpoint locally:

```
curl http://localhost:4000/transaction
```

The screenshot shows a terminal window within the AWS Systems Manager console. The terminal output displays the configuration of a PM2 service named 'hp'. It includes environment variables for PATH, HOME, and PIDFILE, and a command list to enable and start the service. The PM2 daemon is shown saving the process list and performing a health check. The terminal output is as follows:

```
LimitNPROC=infinity
LimitCORE=infinity
Environment=PATH=/home/ec2-user/.nvm/versions/node/v16.20.2/bin:/home/ec2-user/.local/bin:/home/ec2-user/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/var/lib/napd/snap/bin:/home/ec2-user/.nvm/versions/node/v16.20.2/bin:/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
Environment=PM2_HOME=/root/.pm2
PIDFile=/root/.pm2/pm2.pid
Restart=on-failure

ExecStart=/home/ec2-user/.nvm/versions/node/v16.20.2/lib/node_modules/pm2/bin/pm2 resurrect
ExecReload=/home/ec2-user/.nvm/versions/node/v16.20.2/lib/node_modules/pm2/bin/pm2 reload all
ExecStop=/home/ec2-user/.nvm/versions/node/v16.20.2/lib/node_modules/pm2/bin/pm2 kill

[Install]
WantedBy=multi-user.target

Target path
/etc/systemd/system/pm2-ec2-user--hp.service
Command list
[ 'systemctl enable pm2-ec2-user--hp' ]
[PM2] Writing init configuration in /etc/systemd/system/pm2-ec2-user--hp.service
[PM2] Making script booting at startup...
[PM2] [-] Executing: systemctl enable pm2-ec2-user--hp...
Created symlink /etc/systemd/system/multi-user.target.wants/pm2-ec2-user--hp.service -> /etc/systemd/system/pm2-ec2-user--hp.service.
[PM2] [v] Command successfully executed.
+-----+
[PM2] Freeze a process list on reboot via:
$ pm2 save

[PM2] Remove init script via:
$ pm2 unstartup systemd
[ec2-user@ip-10-0-2-105 app-tier]$ pm2 save
[PM2] Saving current process list...
[PM2] Successfully saved in /home/ec2-user/.pm2/dump.pm2
[ec2-user@ip-10-0-2-105 app-tier]$ curl http://localhost:4000/health
"This is the health check"[ec2-user@ip-10-0-2-105 app-tier]$
```

If we see both of these responses, then your networking, security, database and app configurations are correct.

Part 4: Internal Load Balancing and Auto Scaling

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

App Tier AMI

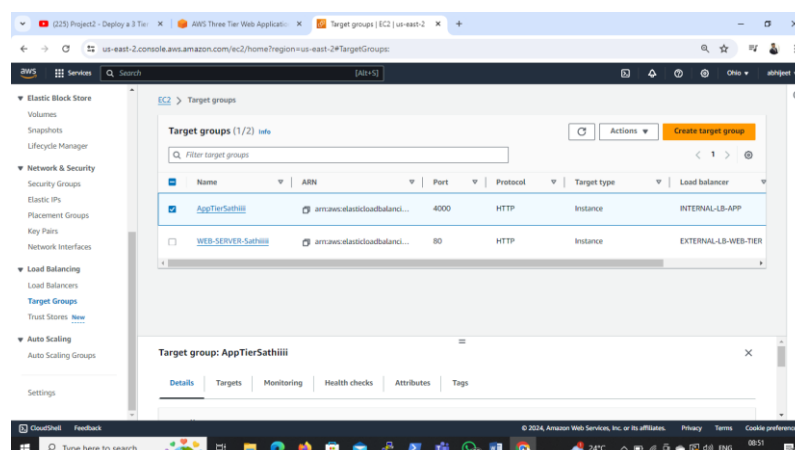
1. Navigate to **Instances** on the left hand side of the EC2 dashboard. Select the app tier instance we created and under **Actions** select **Image and templates**. Click **Create Image**.
2. Give the image a name and description and then click **Create image**. This will take a few minutes, but if you want to monitor the status of image creation you can see it by clicking **AMIs** under **Images** on the left hand navigation panel of the EC2 dashboard.

Target Group

1. While the AMI is being created, we can go ahead and create our target group to use with the load balancer. On the EC2 dashboard navigate to **Target Groups** under **Load Balancing** on the left hand side. Click on **Create Target Group**.
2. 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. Select **Instances** as the target type and give it a name.

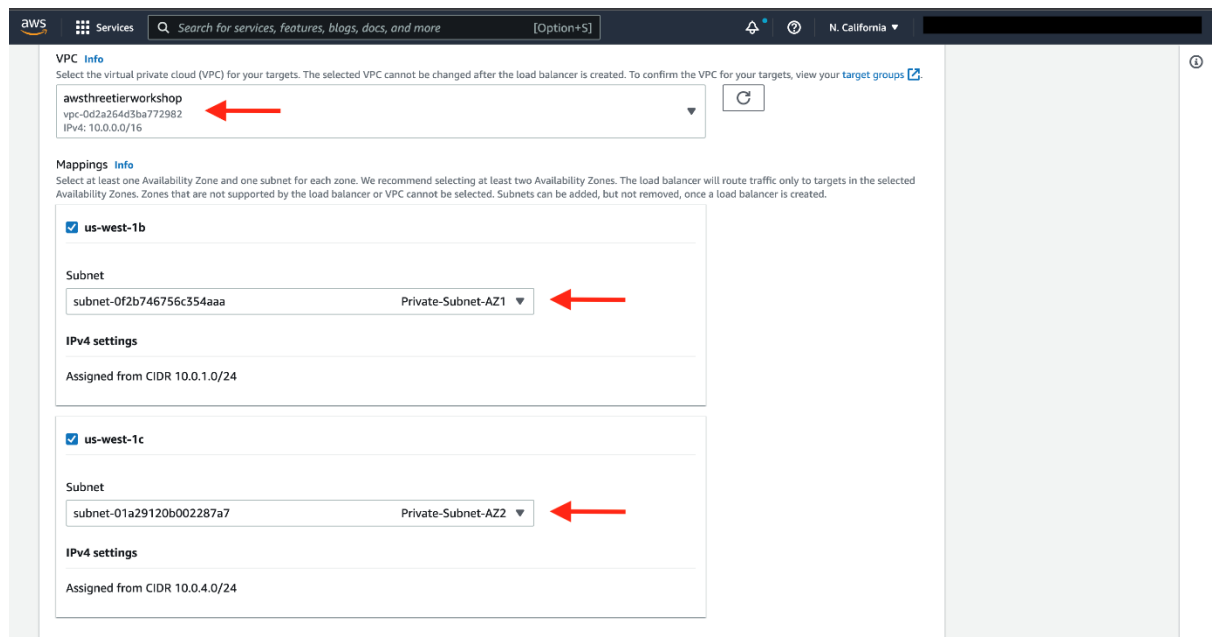
Then, set the protocol to **HTTP** and the port to 4000. Remember that this is the port our Node.js app is running on. Select the VPC we've been using thus far, and then change the health check path to be **/health**. This is the health check endpoint of our app. Click **Next**.

3. We are **NOT** going to register any targets for now, so just skip that step and create the target group.



Internal Load Balancer

1. On the left hand side of the EC2 dashboard select **Load Balancers** under **Load Balancing** and click **Create Load Balancer**.
2. We'll be using an **Application Load Balancer** for our **HTTP** traffic so click the create button for that option.
3. After giving the load balancer a name, be sure to select **internal** since this one will not be public facing, but rather it will route traffic from our web tier to the app tier.
4. Select the correct network configuration for VPC and private subnets.



5. Select the security group we 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 we just created, so select it from the dropdown, and create the load balancer.

Launch Template

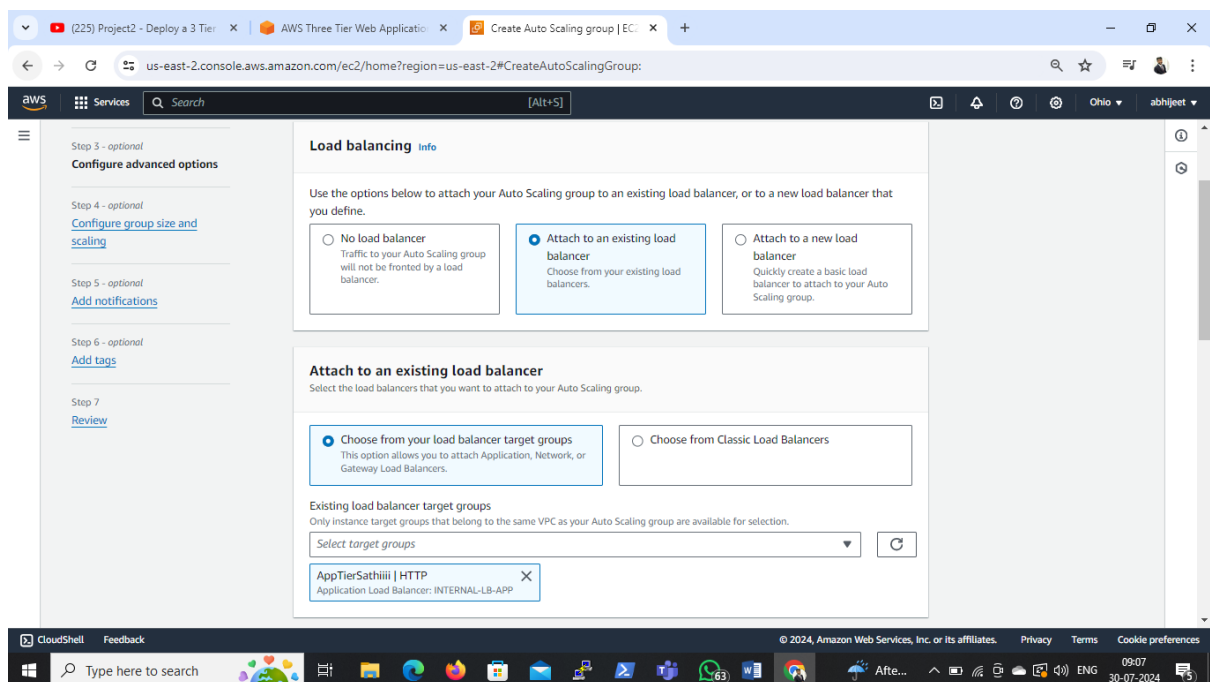
1. Before we configure Auto Scaling, we need to create a Launch template with the AMI we created earlier. On the left side of the EC2 dashboard navigate to **Launch Template** under **Instances** and click **Create Launch Template**.
2. Name the Launch Template, and then under **Application and OS Images** include the app tier AMI you created.

Under **Instance Type** select t2.micro. For **Key pair** and **Network Settings** don't include it in the template. We don't need a key pair to access our instances and we'll be setting the network information in the autoscaling group.

Set the correct security group for our app tier, and then under **Advanced details** use the same IAM instance profile we have been using for our EC2 instances.

Auto Scaling

1. We will now create the Auto Scaling Group for our app instances. On the left side of the EC2 dashboard navigate to **Auto Scaling Groups** under **Auto Scaling** and click **Create Auto Scaling group**.
2. Give your Auto Scaling group a name, and then select the Launch Template we just created and click next.
3. On the **Choose instance launch options** page set your VPC, and the private instance subnets for the app tier and continue to step 3.
4. For this next step, attach this Auto Scaling Group to the Load Balancer we just created by selecting the existing load balancer's target group from the dropdown. Then, click next.



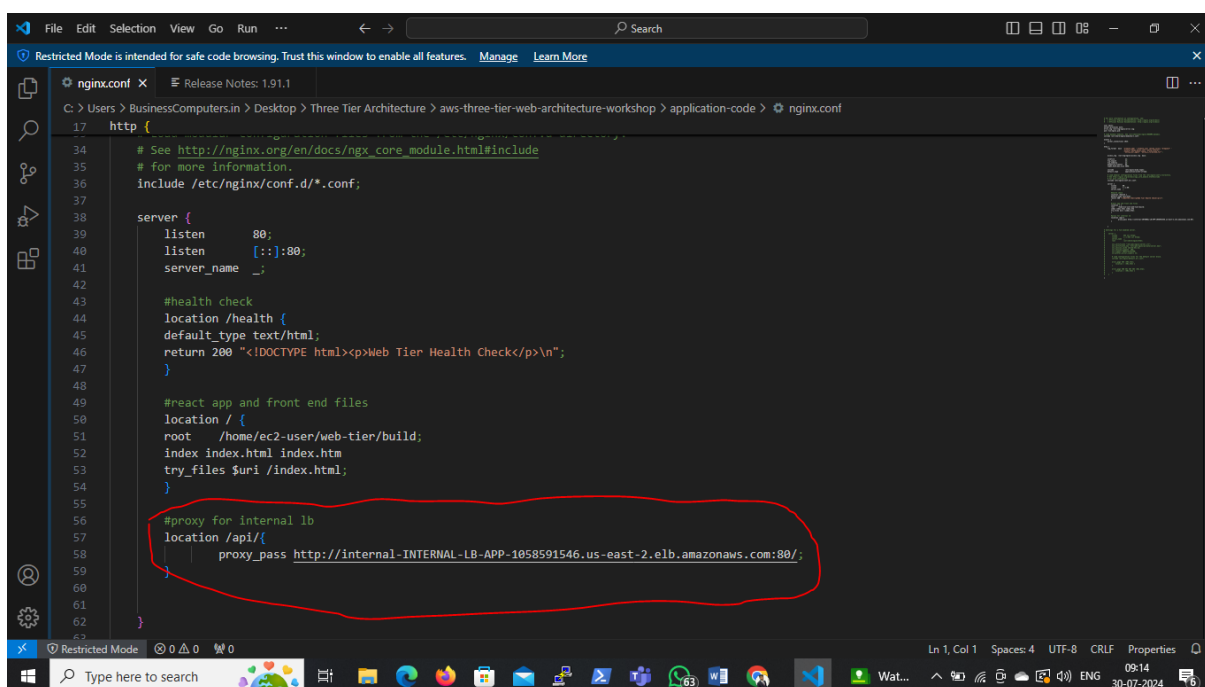
5. For **Configure group size and scaling policies**, set desired, minimum and maximum capacity to **2**. Click skip to review and then Create Auto Scaling Group.
6. You should now have your internal load balancer and autoscaling group configured correctly. You should see the autoscaling group spinning up 2 new app tier instances. If you wanted to test if this is working correctly, you can delete one of your new instances manually and wait to see if a new instance is booted up to replace it.

Part 5: Web Tier Instance Deployment

In this section we will deploy an EC2 instance for the web tier and make all necessary software configurations for the NGINX web server and React.js website.

Update Config File

Before we create and configure the web instances, open up the **application-code/nginx.conf** file from the repo we downloaded. Scroll down to **line 58** and replace **[INTERNAL-LOADBALANCER-DNS]** with our internal load balancer's DNS entry. We can find this by navigating to our internal load balancer's details page.



```
17 http {
34 # See http://nginx.org/en/docs/nginx_core_module.html#include
35 # for more information.
36 include /etc/nginx/conf.d/*.conf;
37
38 server {
39     listen      80;
40     listen      [::]:80;
41     server_name _;
42
43     #health check
44     location /health {
45         default_type text/html;
46         return 200 "<!DOCTYPE html><p>Web Tier Health Check</p>\n";
47     }
48
49     #react app and front end files
50     location / {
51         root    /home/ec2-user/web-tier/build;
52         index  index.html index.htm;
53         try_files $uri /index.html;
54     }
55
56     #proxy for internal lb
57     location /api/{
58         proxy_pass http://internal-INTERNAL-LB-APP-1058591546.us-east-2.elb.amazonaws.com:80/;
59     }
60
61
62 }
```

Then, upload this file **and** the **application-code/web-tier** folder to the s3 bucket you created for this lab.

Web Instance Deployment

1. Follow the same instance creation instructions we used for the App Tier instance in **Part 3: App Tier Instance Deployment**, with the exception of the subnet. We will be provisioning this instance in one of our **public subnets**. Make sure to select the correct network components, security group, and IAM role. **This time, auto-assign a public ip** on the **Configure Instance Details page**. Remember to tag the instance with a name so we can identify it more easily.

Connect to Instance

1. Follow the same steps you used to connect to the app instance and change the user to **ec2-user**. Test connectivity here via ping as well since this instance should have internet connectivity:

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

```
ping 8.8.8.8
```

Configure Web Instance

1. We now need to install all of the necessary components needed to run our front-end application. Again, start 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
```

2. Now we need to download our web tier code from our s3 bucket:

```
cd ~/  
aws s3 cp s3://BUCKET_NAME/web-tier/ web-tier --recursive
```

Navigate to the web-layer folder and create the build folder for the react app so we can serve our code:

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

```
npm install
```

```
npm run build
```

3. NGINX can be used for different use cases like load balancing, content caching etc, but we will be using it as a web server that we 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
```

4. We will now have to configure NGINX. Navigate to the Nginx configuration file with the following commands and list the files in the directory:

```
cd /etc/nginx
```

```
ls
```

You should see an nginx.conf file. We're going to delete this file and use the one we uploaded to s3. Replace the bucket name in the command below with the one you created for this workshop:

```
sudo rm nginx.conf
```

```
sudo aws s3 cp s3://BUCKET_NAME/nginx.conf .
```

Then, restart Nginx with the following command:

```
sudo service nginx restart
```

To make sure Nginx has permission to access our files execute this command:

```
chmod -R 755 /home/ec2-user
```

And then to make sure the service starts on boot, run this command:

```
sudo chkconfig nginx on
```

5. Now when you plug in the public IP of your web tier instance, you should see your website, which you can find on the Instance details page on the EC2 dashboard. If you have the database connected and working correctly, then you will also see the database working. You'll be able to add data. Careful with the delete button, that will clear all the entries in your database.

Part 6: External Load Balancer and Auto Scaling

In this section of the project we will create an Amazon Machine Image (AMI) of the web tier instance we just created, and use that to set up autoscaling with an external facing load balancer in order to make this tier highly available.

Web Tier AMI

1. Navigate to **Instances** on the left hand side of the EC2 dashboard. Select the web tier instance we created and under **Actions** select **Image and templates**. Click **Create Image**.
2. Give the image a name and description and then click **Create image**. This will take a few minutes, but if you want to monitor the status of image creation you can see it by clicking **AMIs** under **Images** on the left hand navigation panel of the EC2 dashboard.

Target Group

1. While the AMI is being created, we can go ahead and create our target group to use with the load balancer. On the EC2 dashboard navigate to **Target Groups** under **Load Balancing** on the left hand side. Click on **Create Target Group**.
2. 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. Select **Instances** as the target type and give it a name.
3. Then, set the protocol to **HTTP** and the port to 80. Remember this is the port NGINX is listening on. Select the VPC we've been using thus far, and then change the health check path to be **/health**. Click **Next**.
4. We are **NOT** going to register any targets for now, so just skip that step and create the target group.

Internet Facing Load Balancer

1. On the left hand side of the EC2 dashboard select **Load Balancers** under **Load Balancing** and click **Create Load Balancer**.
2. We'll be using an **Application Load Balancer** for our **HTTP** traffic so click the create button for that option.
3. After giving the load balancer a name, be sure to select **internet facing** since this one will not be public facing, but rather it will route traffic from our web tier to the app tier.
4. Select the correct network configuration for VPC and **public** subnets.

5. Select the security group we 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 we just created, so select it from the dropdown, and create the load balancer.

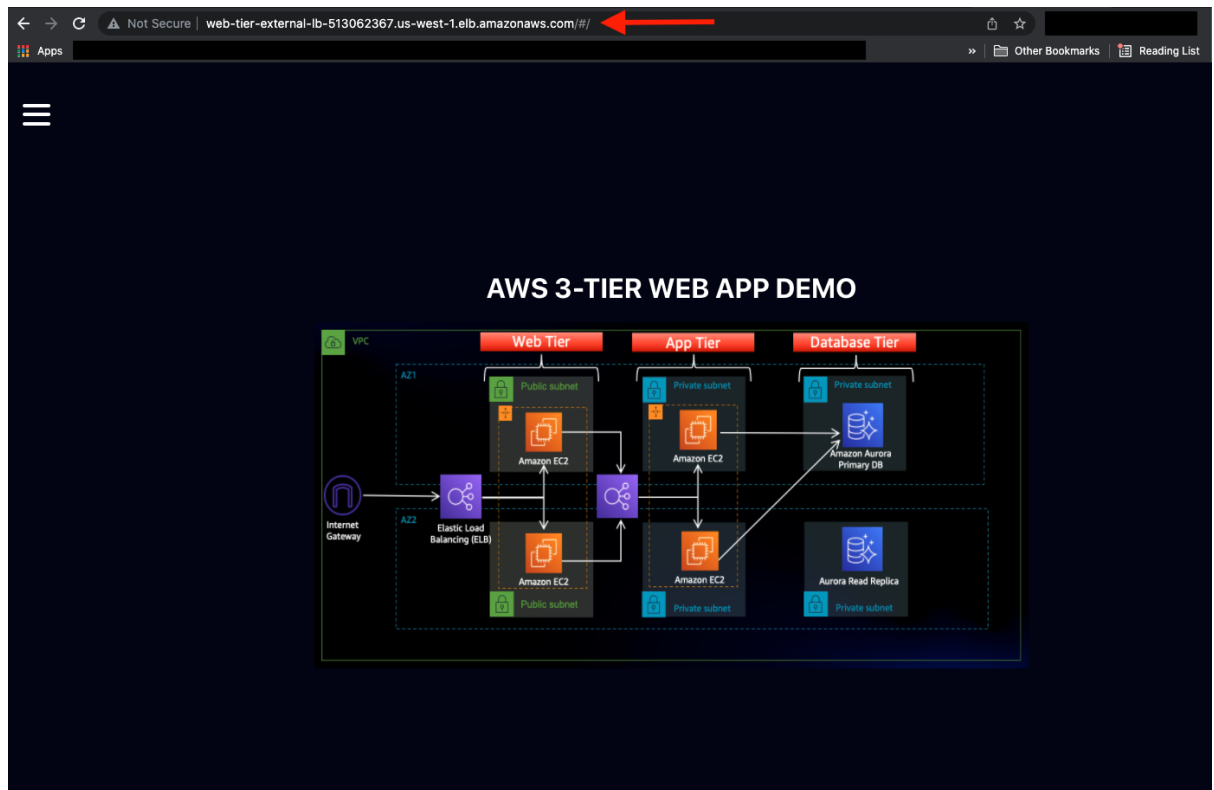
Launch Template

1. Before we configure Auto Scaling, we need to create a Launch template with the AMI we created earlier. On the left side of the EC2 dashboard navigate to **Launch Template** under **Instances** and click **Create Launch Template**.
2. Name the Launch Template, and then under **Application and OS Images** include the app tier AMI you created.
3. Under **Instance Type** select t2.micro. For **Key pair** and **Network Settings** don't include it in the template. We don't need a key pair to access our instances and we'll be setting the network information in the autoscaling group.
4. Set the correct security group for our web tier, and then under **Advanced details** use the same IAM instance profile we have been using for our EC2 instances.

Auto Scaling

1. We will now create the Auto Scaling Group for our web instances. On the left side of the EC2 dashboard navigate to **Auto Scaling Groups** under **Auto Scaling** and click **Create Auto Scaling group**.
2. Give your Auto Scaling group a name, and then select the Launch Template we just created and click next.
3. On the **Choose instance launch options** page set your VPC, and the public subnets for the web tier and continue to step 3.
4. For this next step, attach this Auto Scaling Group to the Load Balancer we just created by selecting the existing web tier load balancer's target group from the dropdown. Then, click next.
5. For **Configure group size and scaling policies**, set desired, minimum and maximum capacity to **2**. Click skip to review and then Create Auto Scaling Group.

We should now have your external load balancer and autoscaling group configured correctly. You should see the autoscaling group spinning up 2 new web tier instances. If we wanted to test if this is working correctly, we can delete one of our new instances manually and wait to see if a new instance is booted up to replace it. To test if our entire architecture is working, navigate to your external facing load balancer, and plug in the DNS name into your browser.



<https://youtube.com/shorts/X4831Gi3VwI?si=yLXeDb7jHv4OJLkW>

