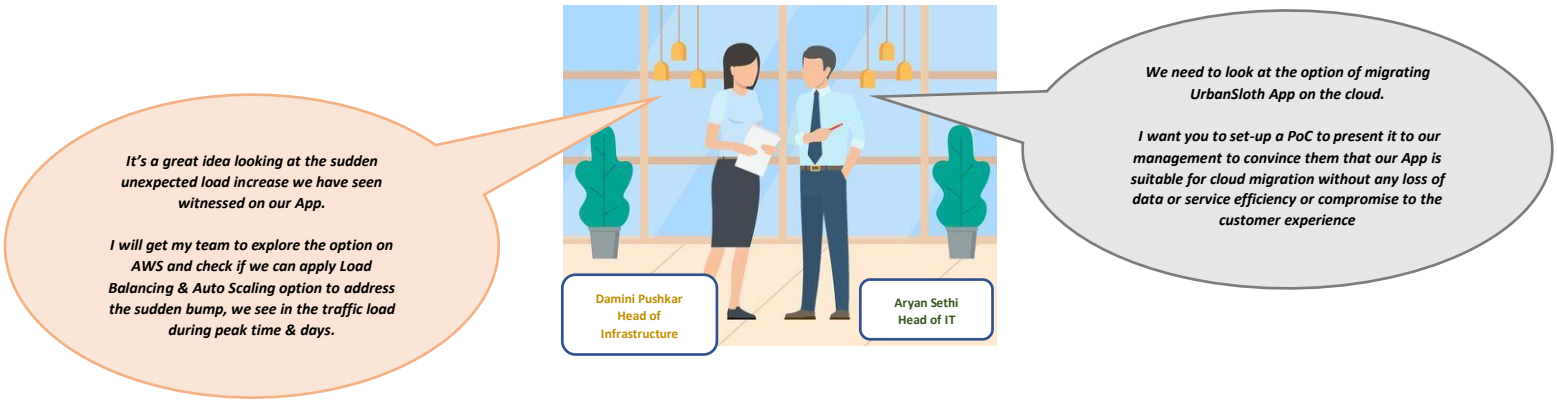
Capstone Project

Submitted by Rajeev Nitnawre

(PGCP CC B-12, Roll No: 24225)

😈😊 Ⓤŕ b ⓐήS L óƬĤ 🍔😳

UrbanSloth is an up-and-coming unicorn in food delivery apps in India due to their ML engines being able to predict on time delivery of food and service quality they were able to penetrate the Indian market in an almost no time. Right now, UrbanSloth holds the 3rd position among the top online food delivery apps in India. With the plan in place to expand their presence amongst other market in South Asia with global launch planned for coming years.



As part of Damini has assigned to me tasks which would make or break the decision to migrate their application on to the cloud. It includes:

- Migrate the on-premise database onto the database server running on the cloud making sure of secure access to it. That is, for instance it need not be accessible from outside the VPC and should be accessible only from the EC2 instance where the Python application is installed.
- On an EC2 instance, replicate all the dependencies and application libraries and configuration from the on-premise server where the Python application is set up.
- Configure the application to run on normal http port instead of testing & development port 5000.
- Set up auto-scaling with load balancer for the above application configuration.

Section 1 - <u>Project Objective or the Problem Statement</u>

- As a rapidly growing popular food delivery service app in India, UranSloth is witnessing huge upsurge in their customer base and huge upsurge in uptake of the service provided by them.
- They are currently in top 3 position among the food delivery app service and have plans for global launch too.
- Their app is currently hosted on a 3rd party rented infrastructure with an on-premise database. Recently, their app has been witnessing unexpected increase in their load which their app isn't capable of managing (peak time traffic).
- Looking at the growth and customer behaviour on their App, they need to have more control over the working efficiency of their App and more control in managing the traffic especially during peak hours & days.
- The solution to tackle these challenges, they cannot depend on the 3rd party and its limited infrastructure capabilities to handle such demands.
- A leading public cloud platform like AWS has solutions to all their problems. The services that AWS cloud platform towards Urbanslouth app issues are:
    - Hosting the food delivery app on the AWS cloud platform
    - Have the on-premise database server migrate to database running on the cloud ensuring secure access, keeping in mind that the database should be accessible to public.
    - Using AWS EC2 instance have all dependencies & app library & configuration moved (replicated)from on-premise to the cloud.
    - Make the app accessible on normal http port for everyone using the app.
    - Able to efficiently manage the huge upsurge in traffic during peak hours and days which can be tailored, programmed and monitored according to plan without any loss of data or latency.
- Eventually, this will also help them in their plan to expand globally without any more time and resource spend separately to expand their infrastructure.

Section-2: <u>Understanding of Requirements</u>

a. The on-premise database will need to be moved seamlessly without any loss of data. RDS service on AWS will be used for the same. For that purpose, we would have to get the AWS EC2 instance running and connect the RDS server using mysql cli to load the data.

   An important aspect to consider here will be that the database should be secured on the cloud and not be accessible from any public IP address.

b. Get an EC2 instance up and running with apache2 web server and get the Flask Python running on the web server by installing the dependencies moved from the on-premise server

c. The Flask application will then need to be made available on standard http port 80 for open public access (it will have been installed earlier on dev & test port 5000)

d. WSGI (Web Server Gateway Interface) will be installed & enabled. It is used to forward requests from apache web server to a backend Python web application. From there, responses will be passed back to the webserver to reply to the requestor. This will now allow us to access the App from the public IP address/ DNS (name of the instance)

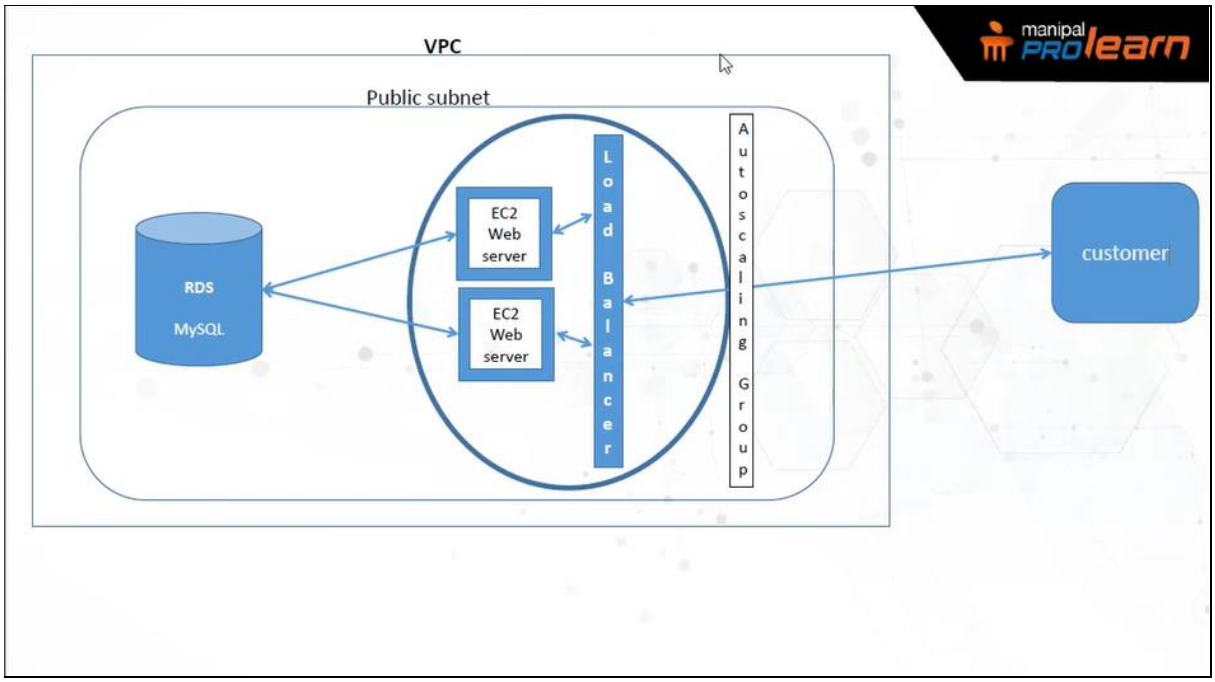
e. To handle the increase/decrease in the load traffic on the App, we will use the Application Load Balancer. This will manage the incoming requests by sending the request in a round robin basis to the various instances created. Once created, the Load balancer will generate its DNS. The earlier enables WSGI configuration setting will be then updated using the Load Balancer DNS.

f. To handle and balance the traffic, we will need to use the Auto Scaling service which will scale out more instances when the average CPU across all instances goes above a set threshold and will scale in when it falls below the threshold. The scaling policies will be set for desired, minimum & maximum capacity as per our requirement.

   For the auto scaling to replicate our instance, we will need to create an AMI – Amazon Machine Image (image of the instance) which the auto scaling will use to create the specified instances to manage and control traffic.

g. Using the DNS name of the load balancer we can now access the web and the traffic will now pass via the Load Balancer which will monitored by the Auto Scaling configuration that we have set to monitor the traffic load.

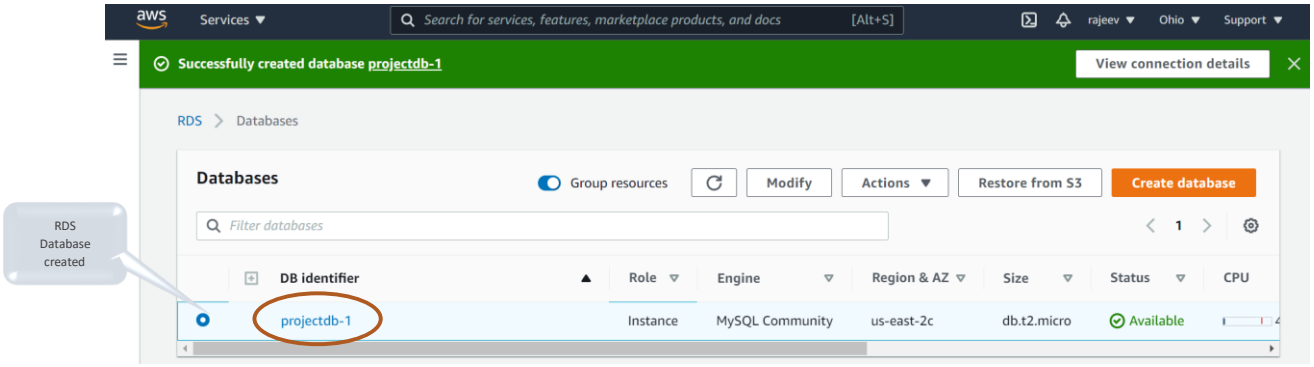
The Basic set-up to be configured will be as follows:

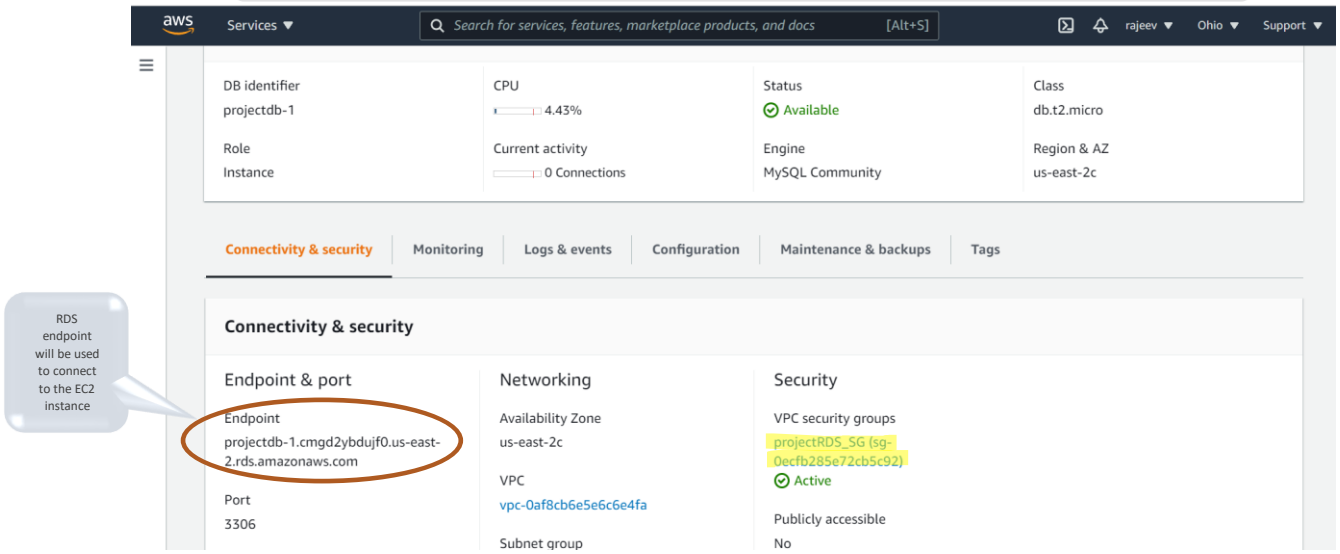


Section-3: Project Implementation

Task 1 - Setting up RDS Database

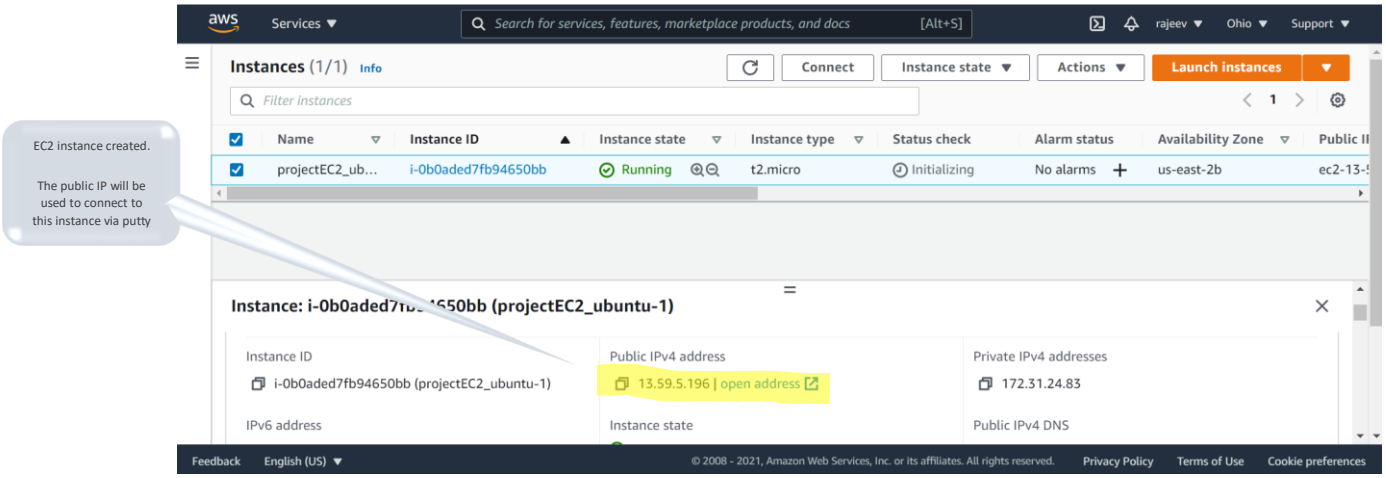
Step-1: Creating the Database



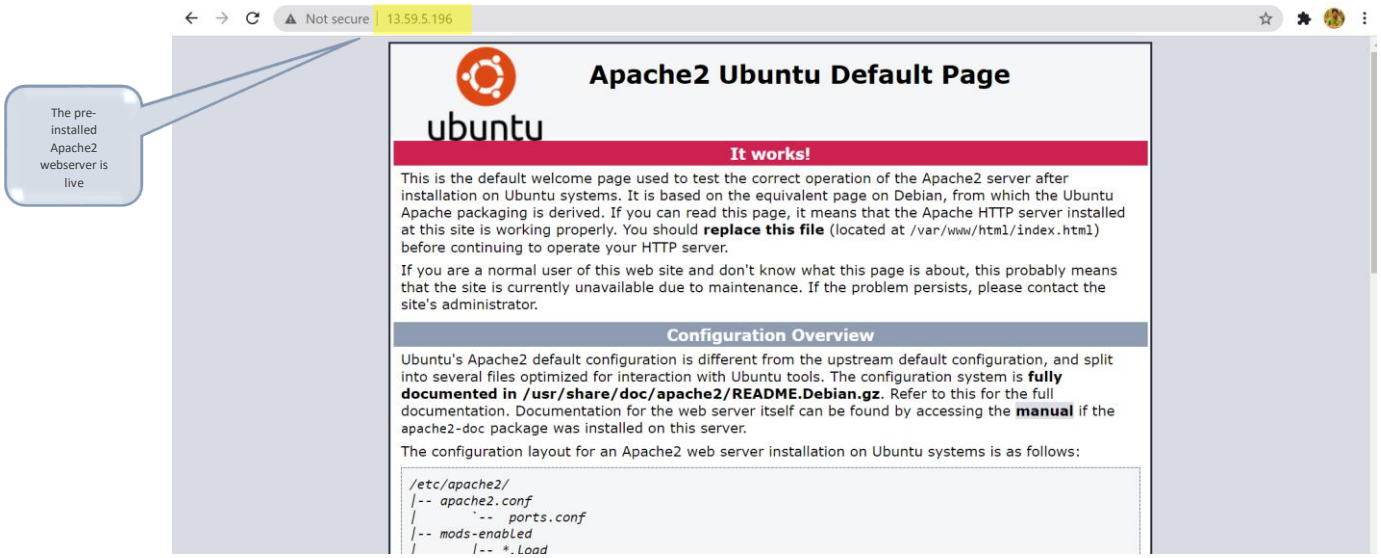


- The endpoint of the RDS database will be used connect to the EC2 instance via mySQL cli
- Security Group created with mySQL for incoming traffic.

Task 2: Launching and Setting up EC2 Instance



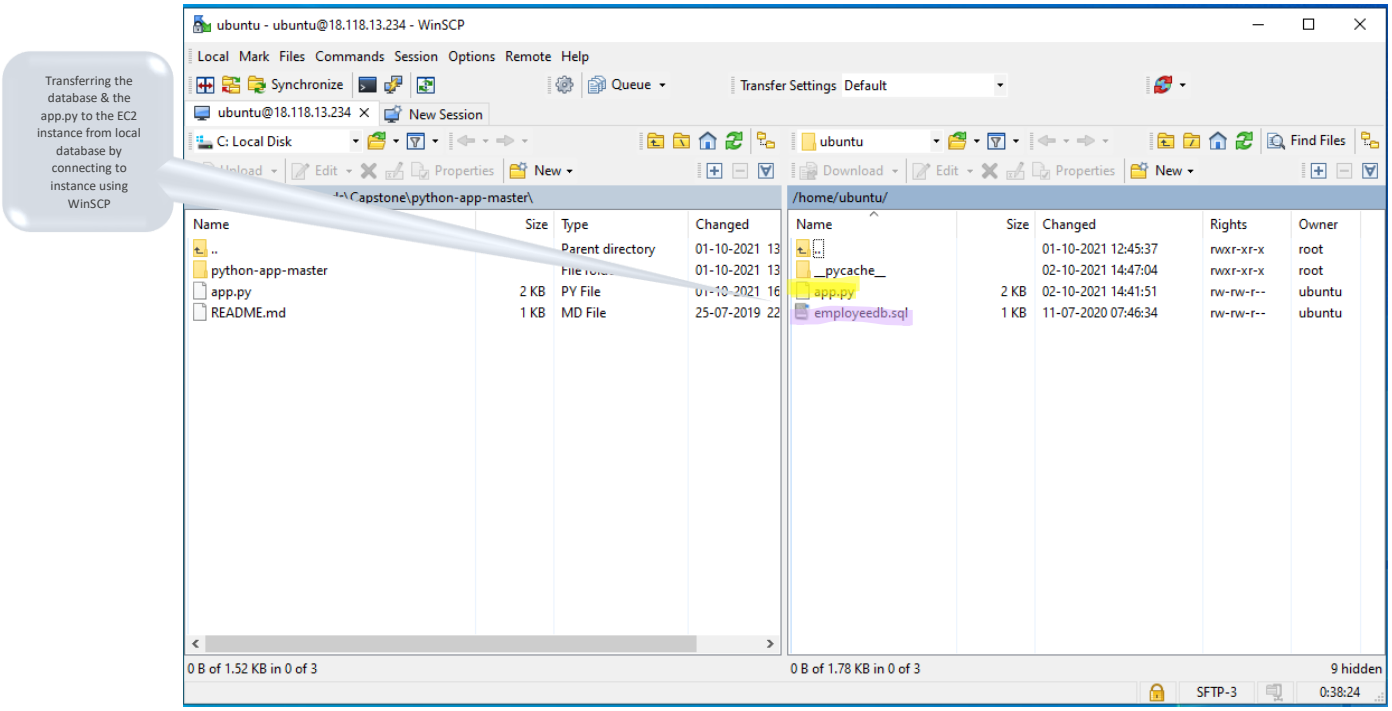
- Ubuntu EC2 instance created by installing apache2 webserver



- Apache webserver reached using the public IP address of the ubuntu instance

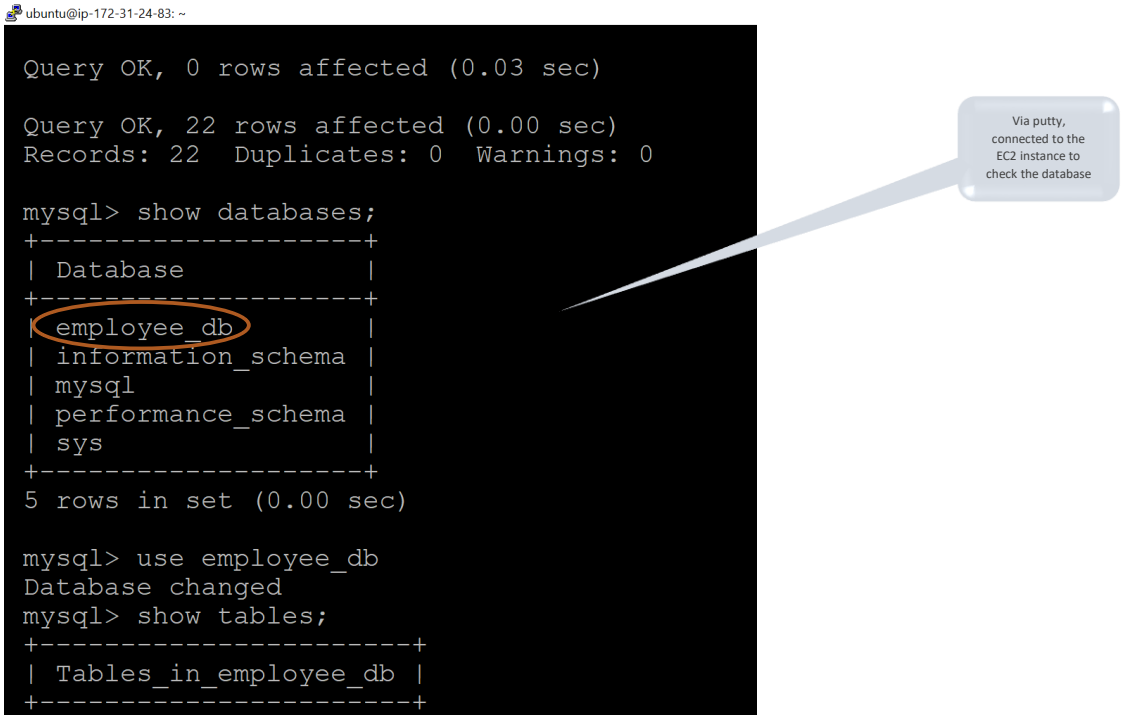
Task 3: Transfer

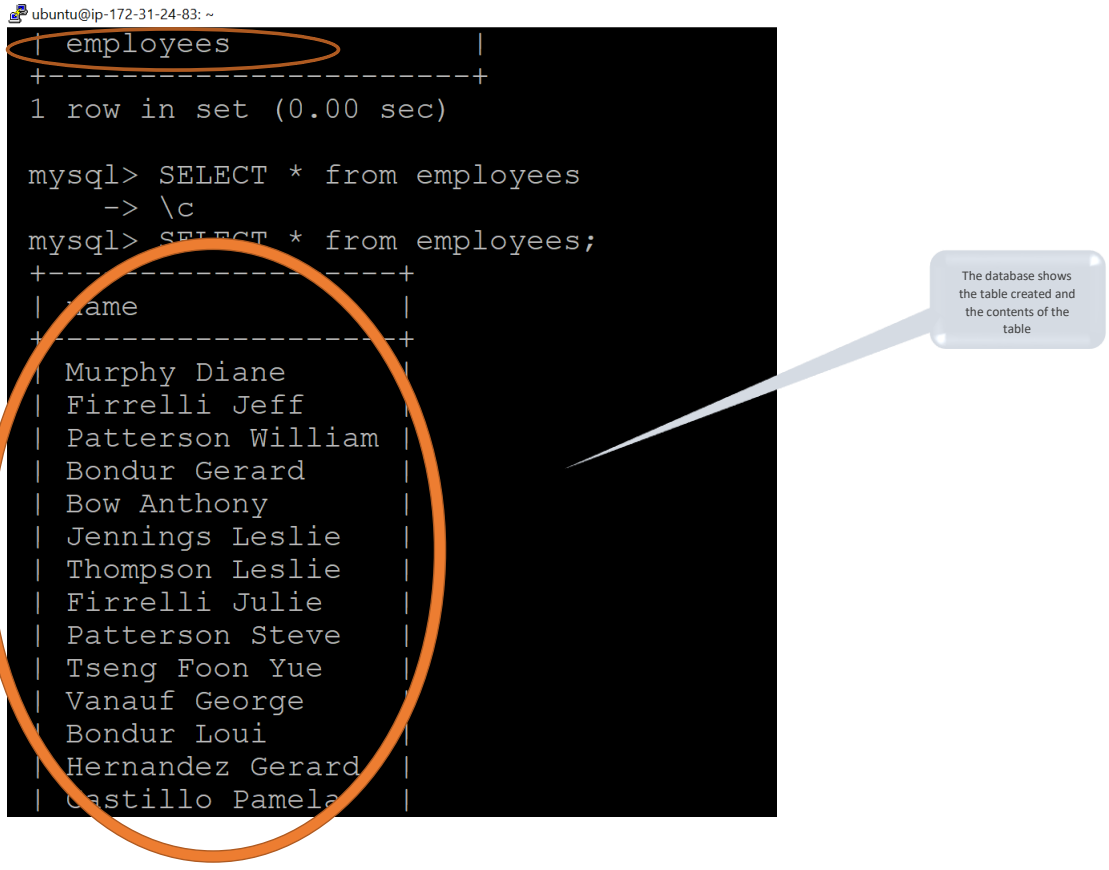
- **database script – *employeedb.sql* to be used to restore or to load the sample database on cloud RDS**
- **test Python web app – *python-app.zip* to be used**



Task 4: Connect to the mySQL RDS instance and check the database that was transferred earlier to the EC2 instance.

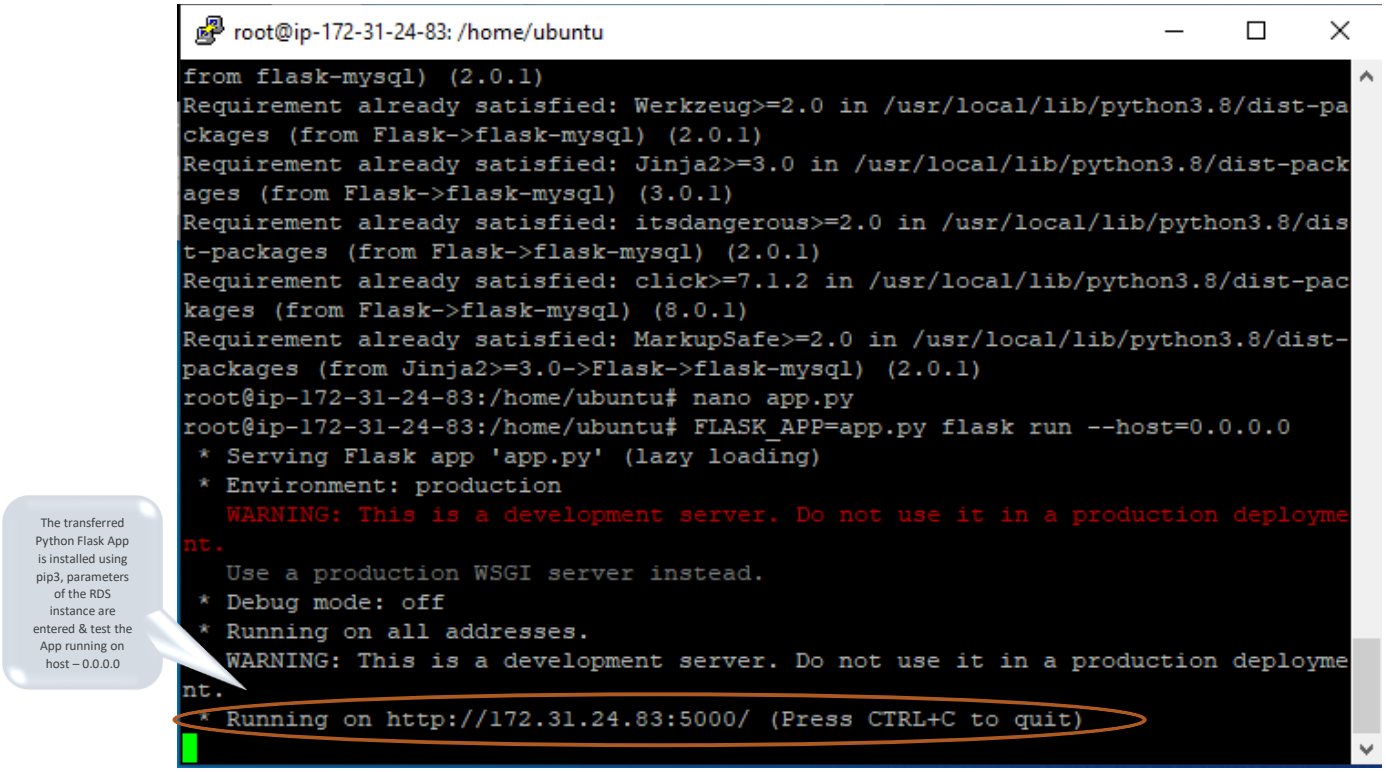
Step 1: look at the table name in the database file and explore the table to check that all the data has been transferred properly.





Task 5: Configuring and testing Python-Flask application

Step 1: Configuring and testing Python-Flask application



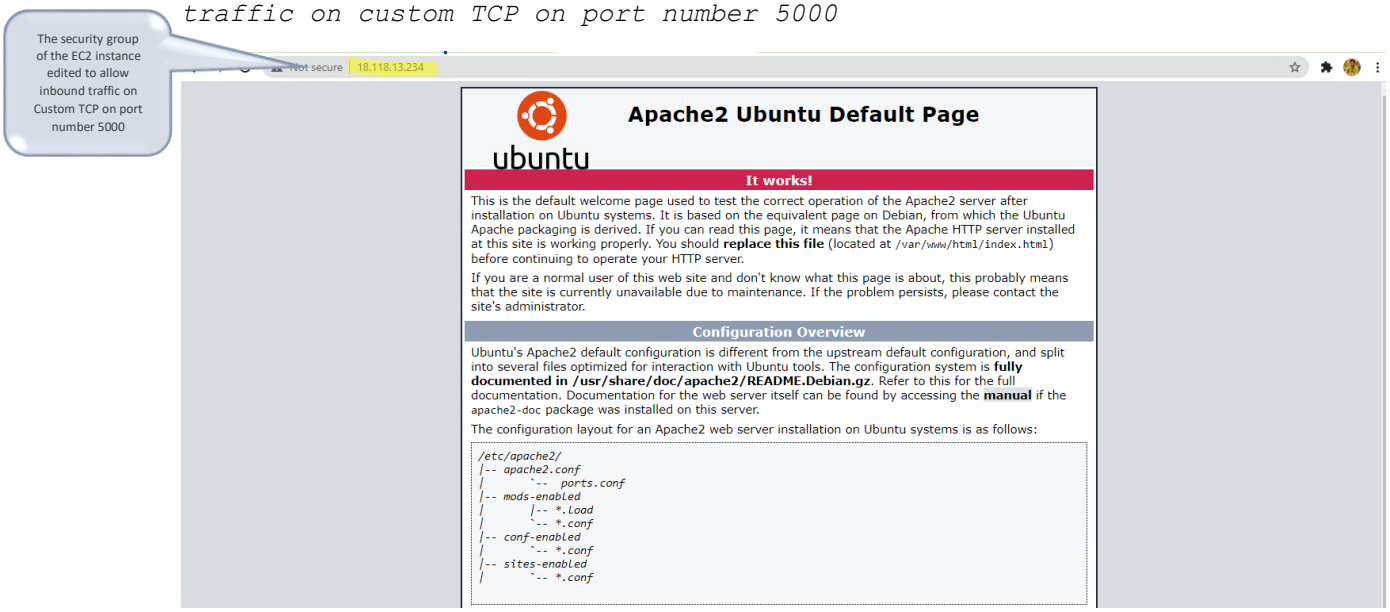
The transferred Python Flask App is installed using pip3, parameters of the RDS instance are entered & test the App running on host – 0.0.0.0

Steps followed towards Configuring and testing Python-Flask application:

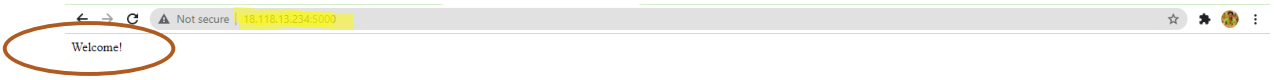
- installing pip3 on the EC2 instance
- installing Flask & flask -mySQL
- changing parameters in the file app.py code to reflect the created RDS instance by putting in appropriate 'user name', 'password' & 'RDS endpoint' – towards the database host.

Step 2: Checking inbound traffic on Custom TCP on the port number 5000

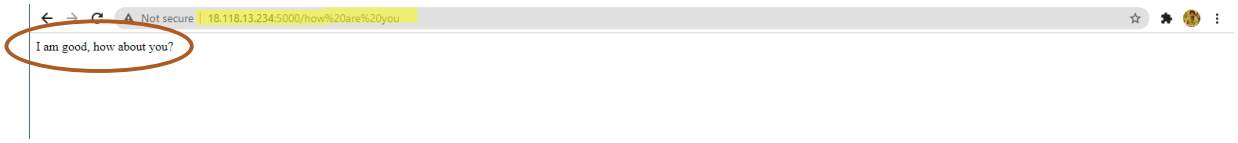
Note: The security group of the EC2 instance was edited to allow inbound traffic on custom TCP on port number 5000



- prior to configuring the Flask application, the apache2 webpage was accessible via the public IP address of the EC2 instance



http://<IP address of your EC2 Instance>:5000



http://<IP address of your EC2 Instance>:5000/how%20are%20you

http://<IP address of your EC2 Instance>:5000/read%20from%20database

This basically shows you can access and use an RDS instance from an EC2 instance through a MySQL CLI client and configure and access through a Flask based Python web application.

Task 6: Install and Enable mod_wsgi (Web Server Gateway Interface Module)

All the following command entered via putty

```
$ sudo apt-get install libapache2-mod-wsgi-py3 python-dev
```

Step 1:

- enable Module wsgi
  `$ sudo a2enmod wsgi`
- setting up a sample flask
  `$ ls -l /var/www`
- create required sub-directories
  `$ sudo mkdir -p /var/www/FlaskApp/FlaskApp`
- create a sample test python application
  `$ sudo cp app.py /var/www/FlaskApp/FlaskApp/__init__.py`

Steps followed to make the Flask application available on the standard http port 80, instead of dev & test port 5000.

Nano editor used to edit & enter the commands.

Step 2:

- configure & enable a New Viirtual Host at port number 80 by creating a configuration file for the FlaskApp

  `$ sudo nano /etc/apache2/sites-available/FlaskApp.conf`

- change the Server Name to the public DNS/IP address of the EC2
- Enable the virtual host with the command line
  `$ sudo a2ensite FlaskApp`
- Activate new configuration
  `systemctl reload apache2`
- Reload apache2
  `$ sudo systemctl reload apache2`

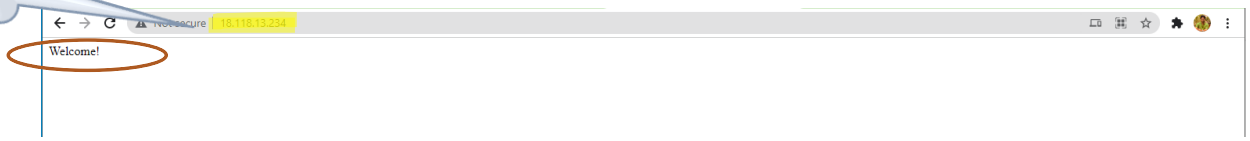
Step 3: Create the .wsgi File

```
$ sudo nano /var/www/FlaskApp/flaskapp.wsgi
```

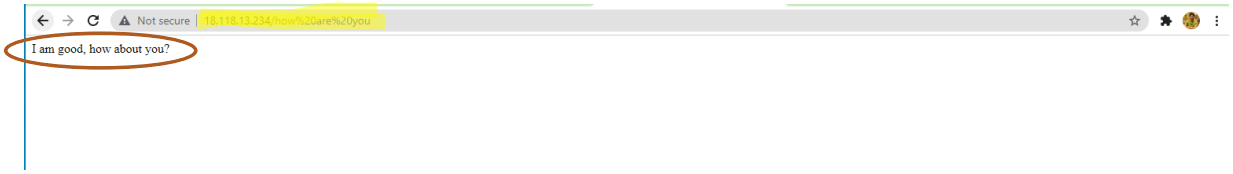
- Restart apache2 server
  `$ sudo service apache2 restart`

The Apache2 server now runs on the public IP/DNS of the EC2 instance

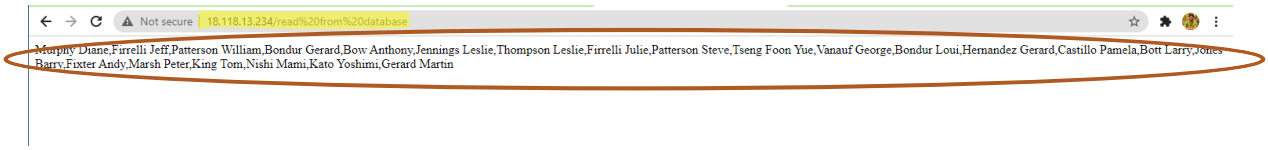
Step 4: Open a browser in the address bar give the Public IP address or Public DNS name of your EC2 instance.



http://<IP address of your EC2 Instance>/

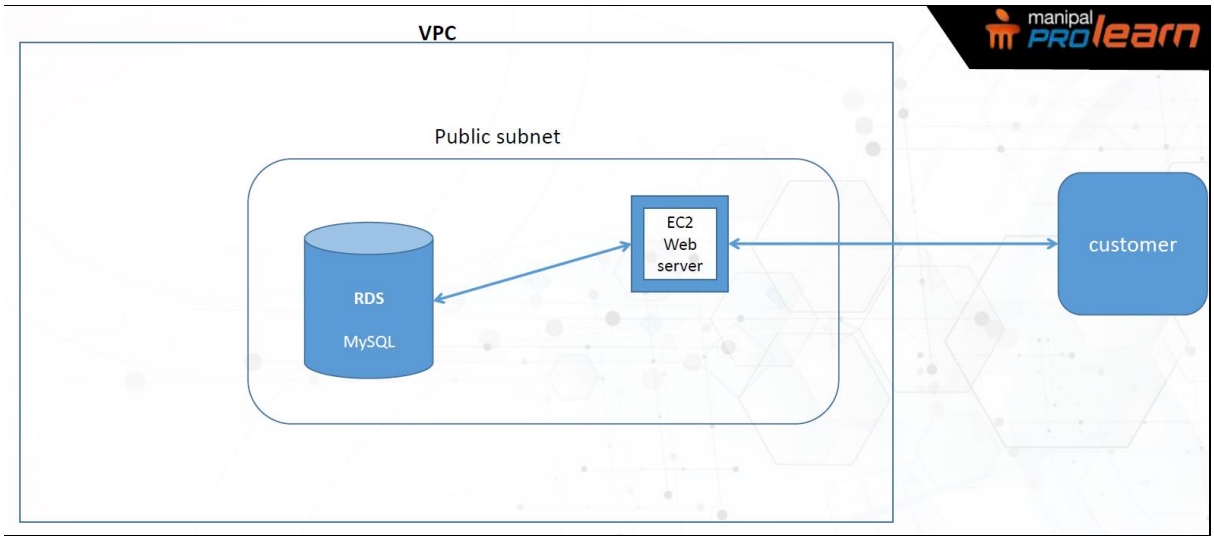


http://<IP address of your EC2 Instance>/how%20are%20you



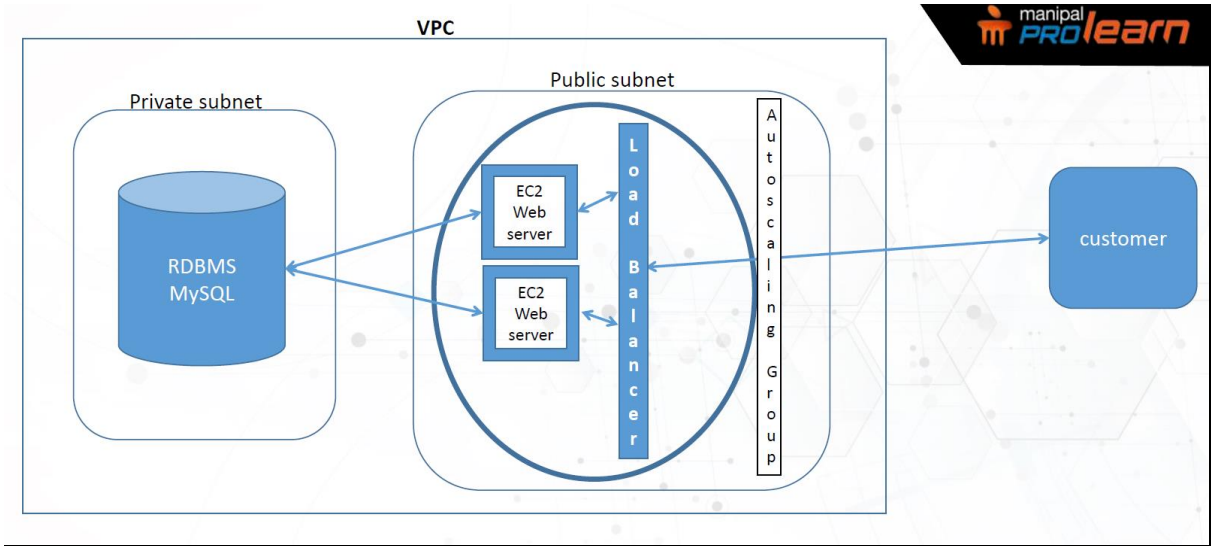
http://<IP address of your EC2 Instance>/read%20from%20database

Task 7: Create a Load Balancer (Application Load Balancer)



Current state

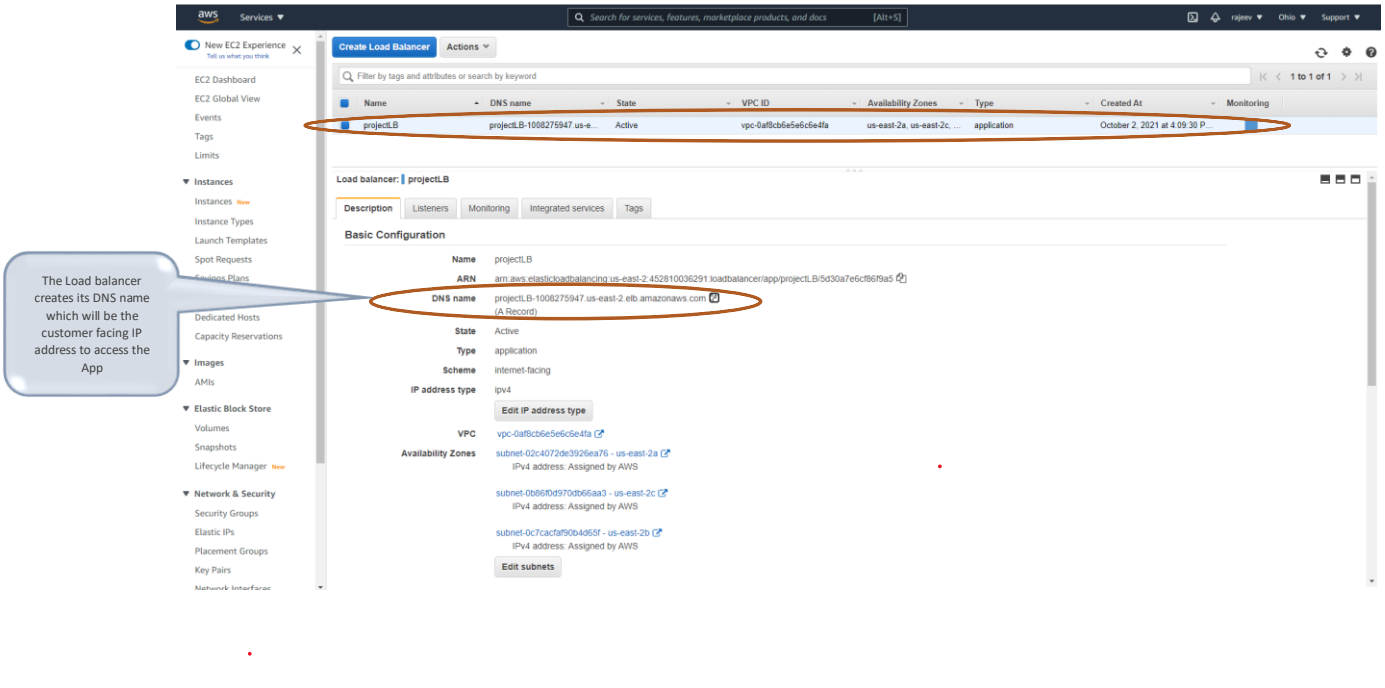
When we put the web application under the Load Balancer the incoming request will come to the Load Balancer instead of the web server directly. The created Load Balancer will generate a DNS which will be used to access the web application for any incoming traffic.



After creating Load Balancer & Auto Scaling Group

Step 1: Create Application Load Balancer

- While creating the Application Load Balancer the scheme set is left to internet-facing (ipv4).
- Listener at http & port 80
- Choose all Available zone and public subnets
- Create new SG
- Select new target Group

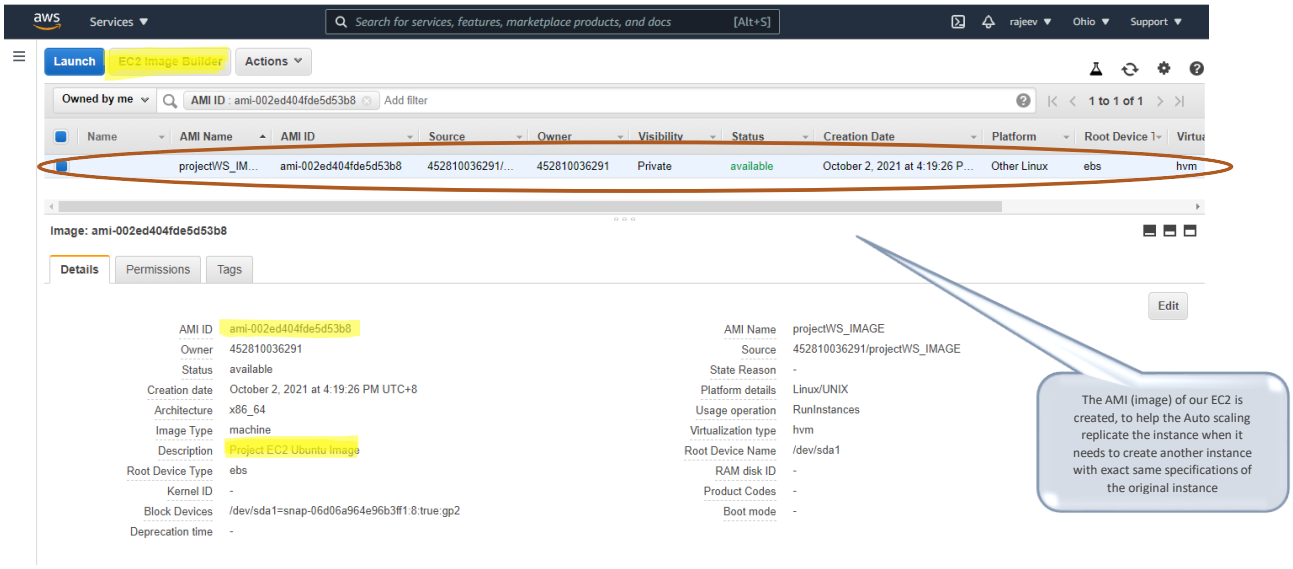


Step 2: Create an EC2 Ubuntu instance and its AMI

- Update the configuration file for the FlaskApp.
  `$ sudo nano /etc/apache2/sites-available/FlaskApp.conf`
- Copy paste the following lines.-->

```
<VirtualHost *:80>
    # Add Public DNS name of your Load Balancer
    ServerName projectLB-1008275947.us-east-2.elb.amazonaws.com
    ServerAdmin anyEMailId@example.com
    # Give an alias to to start your website url with
    WSGIScriptAlias / /var/www/FlaskApp/flaskapp.wsgi
    <Directory /var/www/FlaskApp/FlaskApp/>
        Order allow,deny
        Allow from all
    </Directory>
    ErrorLog ${APACHE_LOG_DIR}/error.log
    LogLevel warn
    CustomLog ${APACHE_LOG_DIR}/access.log combined
</VirtualHost>
```

- Enable the virtual host with the command below
  `$ sudo a2ensite FlaskApp`



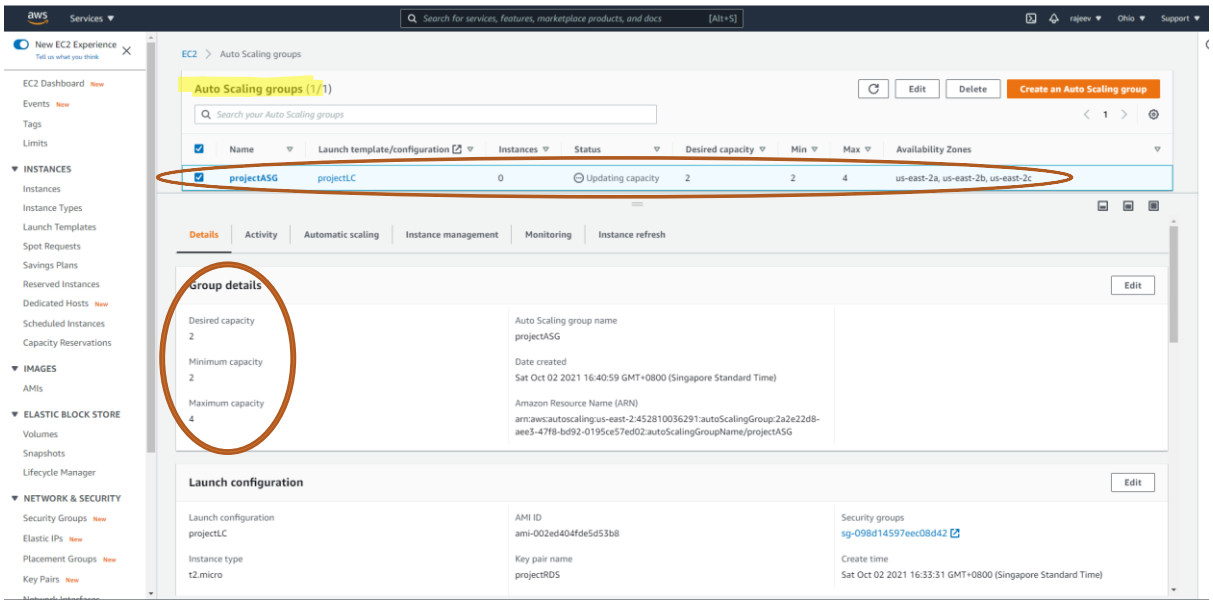
Step 3: Create Launch Configuration

- Choose Create launch configuration
- Choose the created AMI
- Select the hardware configuration t2.micro for the instance.
- Choose the existing security group which is the security created in the load balancer step.

Step 4: Create Auto scaling Group

Configure group size and scaling policies

- Set the Desired Capacity to 2, the Minimum Capacity to 2, and the Maximum Capacity to 4.

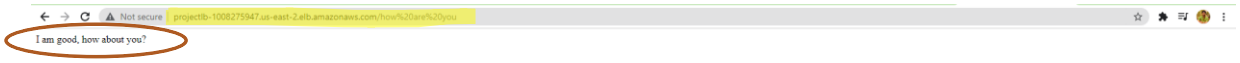


The incoming request now is routed via the Load balancer DNS to the instances. This will enable balancing the traffic incase of heavy load during busy hours/day.

The Auto scaling is enabled which will now monitor the Load balancer load due to

Step 5: Accessing the server using Load balancer DNS

http://<DNS name of your load balancer>/

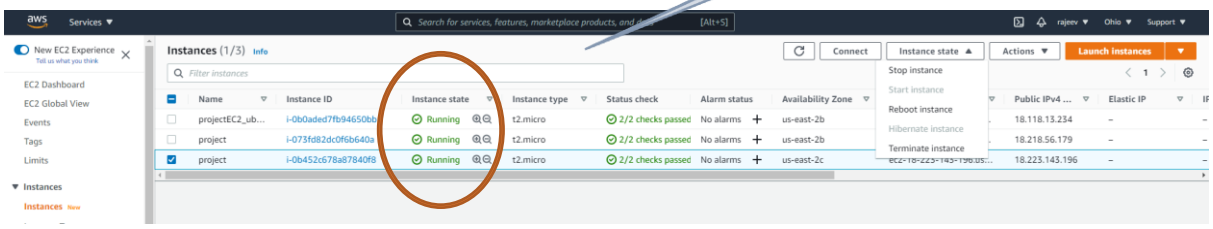


http://<DNS name of your load balancer>/how%20are%20you

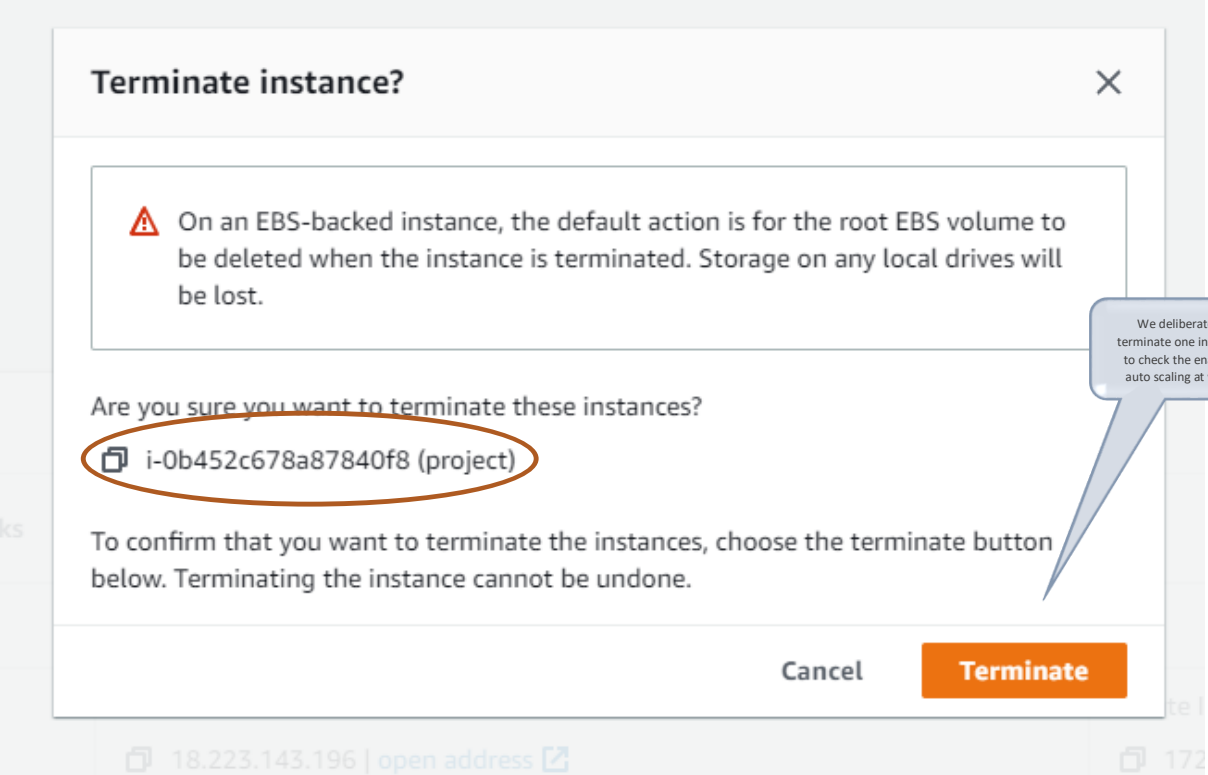
http://<DNS name of your load balancer>/read%20from%20database

We can see 2 images created named "Project" running here. We will now test the enabled auto scaling to see how it works when any instances go down.

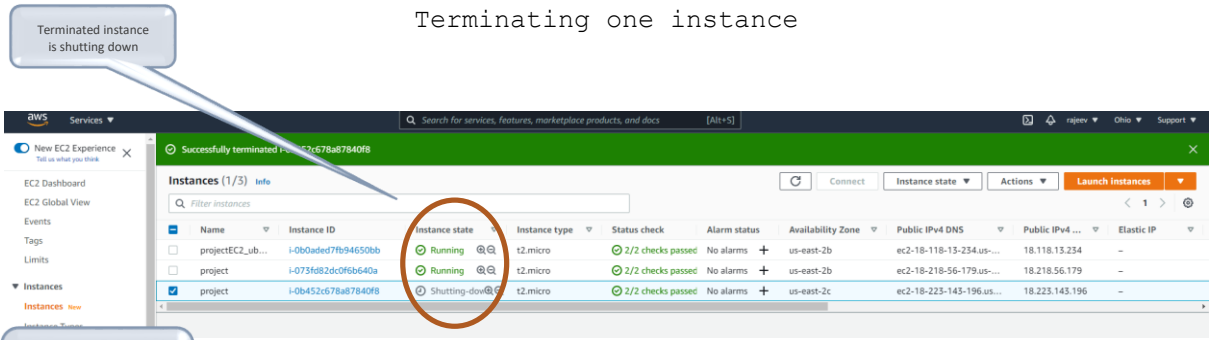
Step 5: Monitor and test the Auto Scaling group



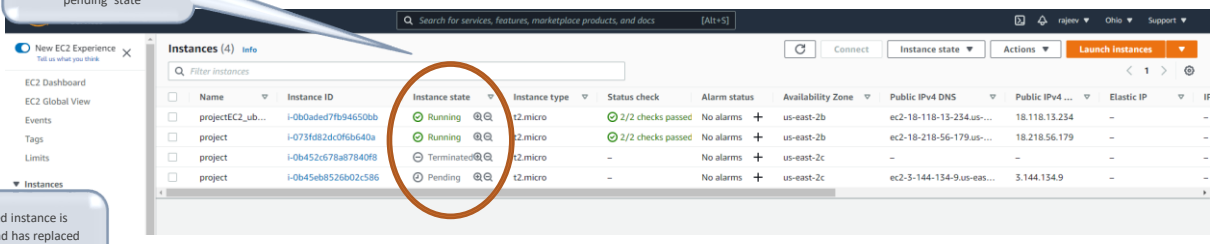
Both instances are currently running. We will deliberately terminate one instance to check if the auto-scaling functionality kicks in



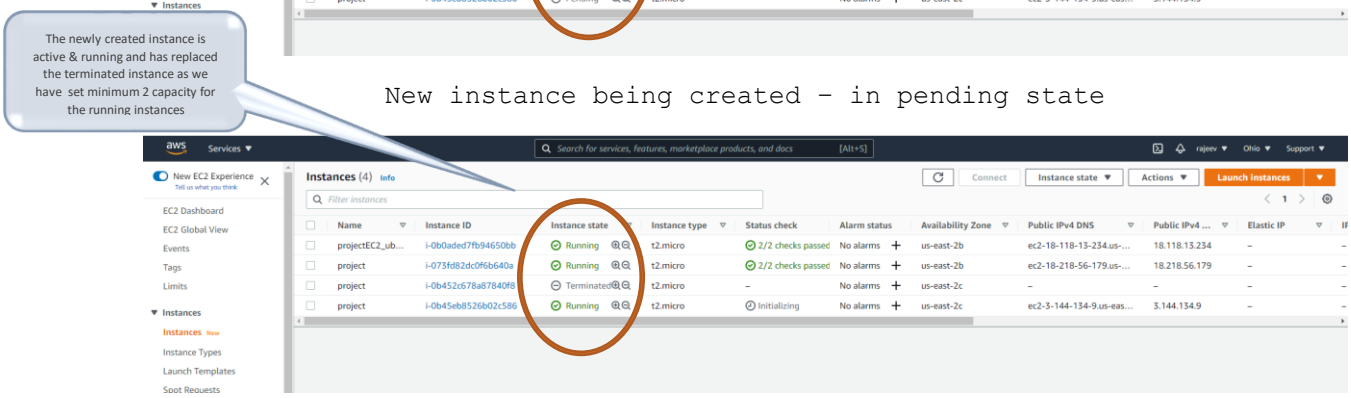
Terminating one instance



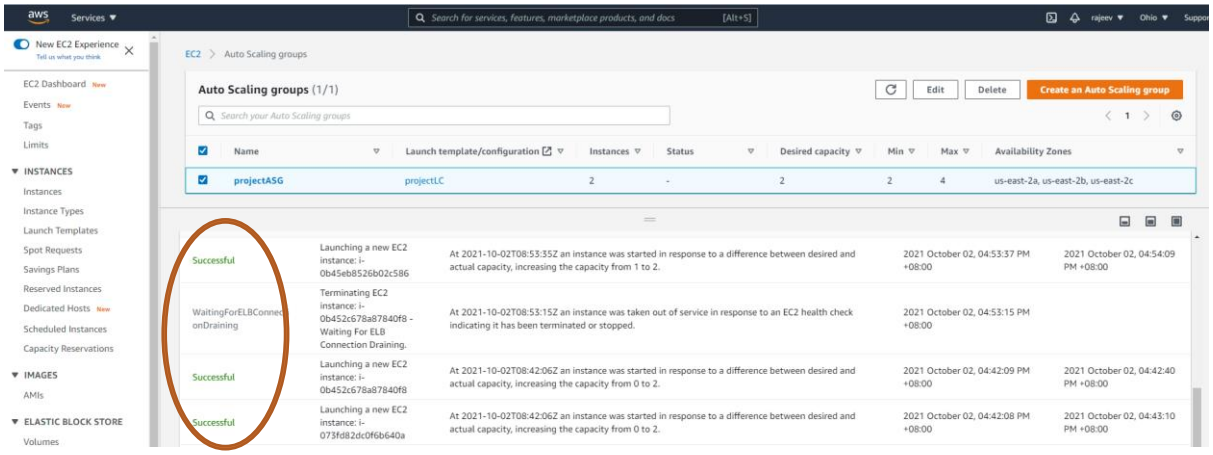
Instance shutting down



New instance being created – in pending state



New instance is up & running while the old one is terminated



Auto scaling functionality works fine

Conclusion:

The setting up of the Proof of Concept (POC) has worked well for the UrbanSloth App. We were able to

- Host the food delivery app on the AWS cloud platform
- Migrate the on-premise database server to the cloud ensuring secure access.
- The dependencies app library & configuration were moved (replicated) from on-premise to the cloud.
- App was made accessible on normal http port for everyone using the app.
- Using Load balancing & configuring Auto Scaling we were able to successfully manage the scaling (up/down) of the server to manage the load as per traffic.