

Tab 1



AWS CAPESTONE PROJECT

Project overview

This project demonstrates the **deployment of a production-grade Django web application** on **Amazon Web Services (AWS)**. The primary focus is not on building new application features but on designing, provisioning, and operating a **robust, scalable, secure, and observable infrastructure** suitable for real-world workloads.

The project leverages core AWS services to create a fully managed environment optimized for availability, performance, and security. It ensures the Django app is **deployed using EC2 instances**, automatically scaled with **Auto Scaling Groups**, and load-balanced through an **Application Load Balancer**. The database is hosted on **Amazon RDS**, while **Amazon S3** is used to store application logs. **IAM roles** are used for secure access between services without hardcoded credentials.

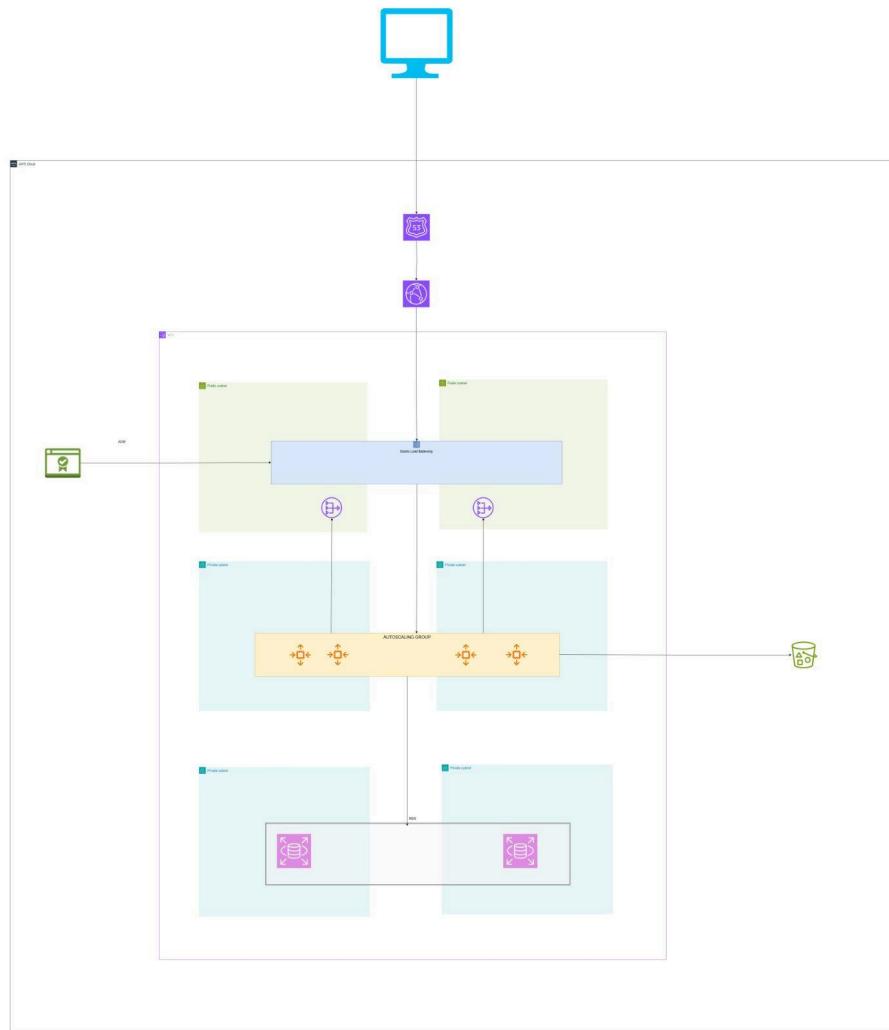
Custom domain routing is handled using **Amazon Route 53**, secured via **AWS Certificate Manager (ACM)** for HTTPS. Monitoring, logging, and alerting are achieved using **Amazon CloudWatch** and **SNS**. This setup reflects **real-world infrastructure practices**, with automated cron jobs for pushing logs to S3 and **CloudFront** used to reduce content delivery latency.

Project specification

- This project focuses on deploying a **Django web application** into a **production-ready AWS cloud infrastructure**, with a strong emphasis on scalability, security, and real-time performance.
- The Django application is hosted on **Amazon EC2** instances, running **Gunicorn** as the application server and **Nginx** as a reverse proxy. **Nginx** also handles the serving of static files directly from the server
- All EC2 instances are deployed within a **custom Amazon VPC**, inside **private subnets** for added security. Public subnets are used only for the **Application Load Balancer (ALB)**, which routes traffic securely to the EC2 instances.

- 
- To ensure high availability and handle traffic surges, an **Auto Scaling Group (ASG)** dynamically adjusts the number of EC2 instances based on real-time metrics like CPU utilization.
 - The application connects to **Amazon RDS (MySQL)**, a fully managed, scalable, and fault-tolerant relational database service, for its backend data needs.
 - All application **logs are pushed periodically to Amazon S3** using a scheduled **cron job** running on each EC2 instance. This allows for centralized, durable, and cost-effective log storage, useful for auditing, analysis, and troubleshooting.
 - A **custom domain** is managed through **Amazon Route 53**, which maps the domain to the ALB. **AWS Certificate Manager (ACM)** is used to issue and manage SSL certificates, enabling secure HTTPS traffic.
 - **Amazon CloudWatch** is used to monitor infrastructure and application metrics. Combined with **CloudWatch Logs** and **CloudWatch Alarms**, real-time alerts are sent via **Amazon SNS** for proactive monitoring and response.

System Architecture Diagram



Project Goals

- Host a Django app with a secure, production-grade infrastructure
- Ensure auto-scaling for high availability
- Use IAM roles and VPC endpoints for secure log storage in S3
- Monitor health and performance using CloudWatch and alerting via SNS
- Apply SSL and route traffic through a custom domain using Route 53 and ACM

AWS Service Configuration

Step 1 : VPC and subnet configuration

We begin by setting up a **Virtual Private Cloud (VPC)** as the foundational network infrastructure for the application deployment. The VPC is configured with a **total of six subnets**, distributed as follows:

- **2 Public Subnets:** These subnets are used to host the **Application Load Balancer (ALB)**, which acts as the public entry point to the application.
- **4 Private Subnets:** These subnets are used to deploy internal components to ensure high security. This includes:
 - **Auto Scaling Group (ASG):** Hosts the EC2 instances running the Django application backend.
 - **Amazon RDS for MySQL:** A managed relational database service for data storage.

The EC2 instances deployed in the private subnets **do not have direct internet access**. Instead, all incoming traffic is routed through the **Application Load Balancer**, which then forwards it securely to the instances.

To allow the EC2 instances in the private subnets to access the internet (for tasks like software updates, etc.), a **NAT Gateway** is deployed in one of the public subnets. The NAT Gateway allows **outbound internet access** for instances in the private subnets while keeping them isolated from direct inbound traffic from the internet.

Additionally, a **Gateway VPC Endpoint** is configured to allow private connectivity from the EC2 instances (within the private subnets) to **Amazon S3** without traversing the public internet. This setup enhances security by ensuring log data generated by the application can be securely stored in S3.



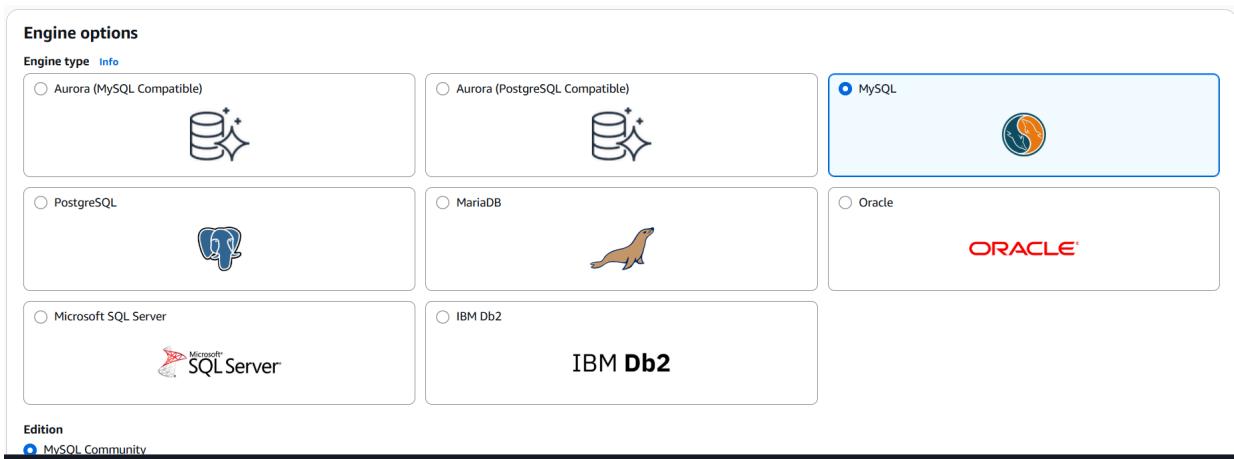
The above diagram illustrates the architectural layout of the VPC and its associated resources.

Step 2: RDS Configuration for MySQL

We are using **Amazon RDS** to create a managed **MySQL database** for our Django application. This setup is designed for a production-grade environment.

Steps to Create the RDS Instance:

- Go to the **AWS Management Console** → search and open **RDS** → click “**Create database**”.
- **Engine selection:**
 - Choose **MySQL** as the database engine.
 - Select the **latest version** (e.g., MySQL 8.0).



- **Database creation method:**
 - Choose **Standard Create**.

Create database Info

Choose a database creation method

Standard create

You set all of the configuration options, including ones for availability, security, backups, and maintenance.

Easy create

Use recommended best-practice configurations. Some configuration options can be changed after the database is created.

- Use the **Production template** for high availability and better performance.

Templates

Choose a sample template to meet your use case.

Production

Use defaults for high availability and fast, consistent performance.

Dev/Test

This instance is intended for development use outside of a production environment.

Sandbox

To develop new applications, test existing applications, or gain hands-on experience with Amazon RDS.

Availability and durability

- **Settings:**

- Set **DB instance identifier** (e.g., sample-prod-db).
- Create a **Master username** and strong password. Save credentials securely.

Settings

DB instance identifier Info

Type a name for your DB instance. The name must be unique across all DB instances owned by your AWS account in the current AWS Region.

sample-prod-db

The DB instance identifier is case-insensitive, but is stored as all lowercase (as in "mydbinstance"). Constraints: 1 to 63 alphanumeric characters or hyphens. First character must be a letter. Can't contain two consecutive hyphens. Can't end with a hyphen.

▼ **Credentials Settings**

Master username Info

Type a login ID for the master user of your DB instance.

admin

1 to 16 alphanumeric characters. The first character must be a letter.

Credentials management

You can use AWS Secrets Manager or manage your master user credentials.

Managed in AWS Secrets Manager - most secure

RDS generates a password for you and manages it throughout its lifecycle using AWS Secrets Manager.

Self managed

Create your own password or have RDS create a password that you manage.

ⓘ If you manage the master user credentials in AWS Secrets Manager, additional charges apply. See [AWS Secrets Manager pricing](#). Additionally, some RDS features aren't supported. See limitations [here](#).

Select the encryption key Info

You can encrypt using the KMS key that Secrets Manager creates or a customer managed KMS key that you create.

aws/secretsmanager (default)



Add new key ⓘ

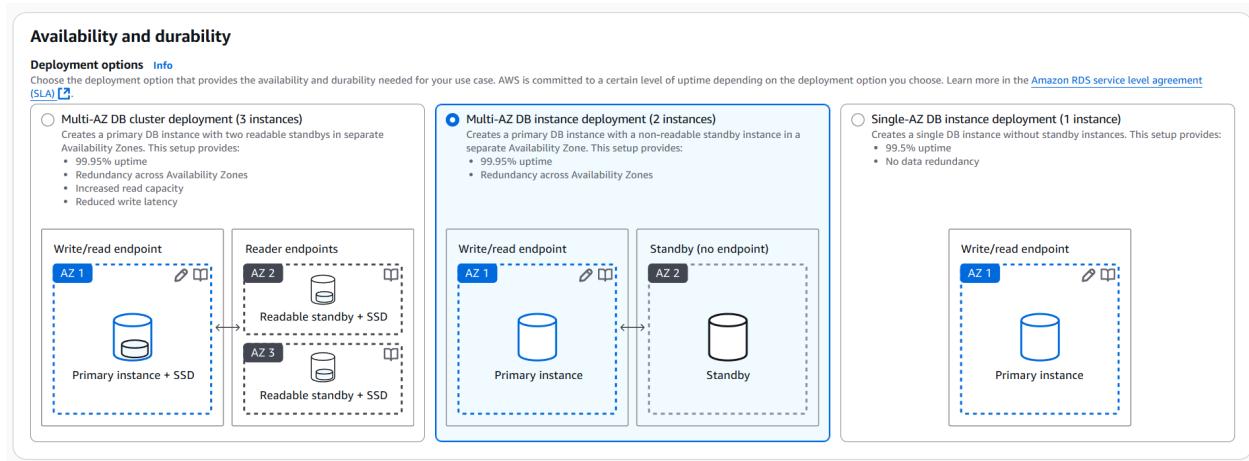
- **Instance specifications:**

- Select instance class (e.g., db . t3 . medium).

- Enable **storage autoscaling** and set **initial storage** (e.g., 20 GB).

- **Availability & durability:**

- Enable **Multi-AZ deployment** for fault tolerance.
- Turn on **automatic backups** (e.g., 7-day retention).



- **Connectivity:**

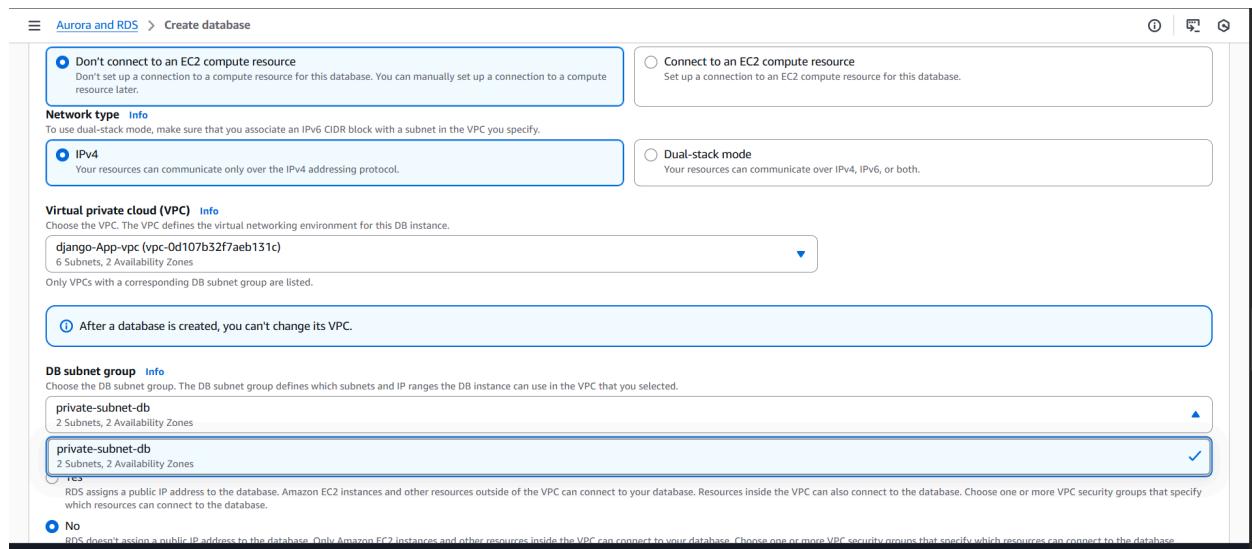
- Select the same **VPC** used for your application servers.

- Use a **private subnet group** to restrict internet access.

The screenshot shows the AWS Aurora and RDS Subnet groups interface. On the left, there's a sidebar with links like Dashboard, Databases, Query editor, Performance insights, Snapshots, Exports in Amazon S3, Automated backups, Reserved instances, and Proxies. Under Subnet groups, it lists Parameter groups, Option groups, Custom engine versions, and Zero-ETL integrations. The main area is titled "Subnet groups (0)" and contains a search bar and a table header with columns for Name, Description, Status, and VPC. A message says "No db subnet groups. You don't have any db subnet groups." with a "Create DB subnet group" button.

The screenshot shows the "Add subnets" wizard. In the "Availability Zones" section, "us-east-1a" and "us-east-1b" are selected. In the "Subnets" section, two subnets are selected: "django-App-subnet-private3-us-east-1a" and "django-App-subnet-private4-us-east-1b". A note at the bottom states: "For Multi-AZ DB clusters, you must select 3 subnets in 3 different Availability Zones." Below this, a table titled "Subnets selected (2)" shows the details of the selected subnets.

Availability zone	Subnet name	Subnet ID	CIDR block
us-east-1a	django-App-subnet-private3-us-east-1a	subnet-0d9d45b800aba9b37	10.0.160.0/20
us-east-1b	django-App-subnet-private4-us-east-1b	subnet-01d657396a8eed827	10.0.176.0/20



- Configure a **security group** to allow port **3306** only from your EC2 instances.
- **Authentication:**
 - Choose **password authentication** for now.
 - IAM authentication can be added optionally.
- **Launch the database:**
 - Click **Create Database** and wait until the status becomes **Available**.
 - Copy the **endpoint URL** for use in your Django application's database settings.

The screenshot shows the AWS RDS console with the following details:

- Summary:**
 - DB identifier: database-1
 - Status: Backing-up
 - Role: Instance
 - Engine: MySQL Community
 - Region & AZ: us-east-1b
- Connectivity & security (selected):**
 - Endpoint:** database-1.c0rgiwca00s9.us-east-1.rds.amazonaws.com
 - Port:** 3306
 - Networking:**
 - Availability Zone: us-east-1b
 - VPC: django-App-vpc (vpc-0d107b32f7aeb131c)
 - Subnet group: private-subnet-db
 - Subnets:
 - subnet-01d657396a8eed827
 - subnet-0d9d45b800aba9b37
 - Security:**
 - VPC security groups: default (sg-093ae5e5a176522d) - Active
 - Publicly accessible: No
 - Certificate authority: rds-ca-rsa2048-g1
 - Certificate authority date: May 26, 2061, 05:04 (UTC+05:30)

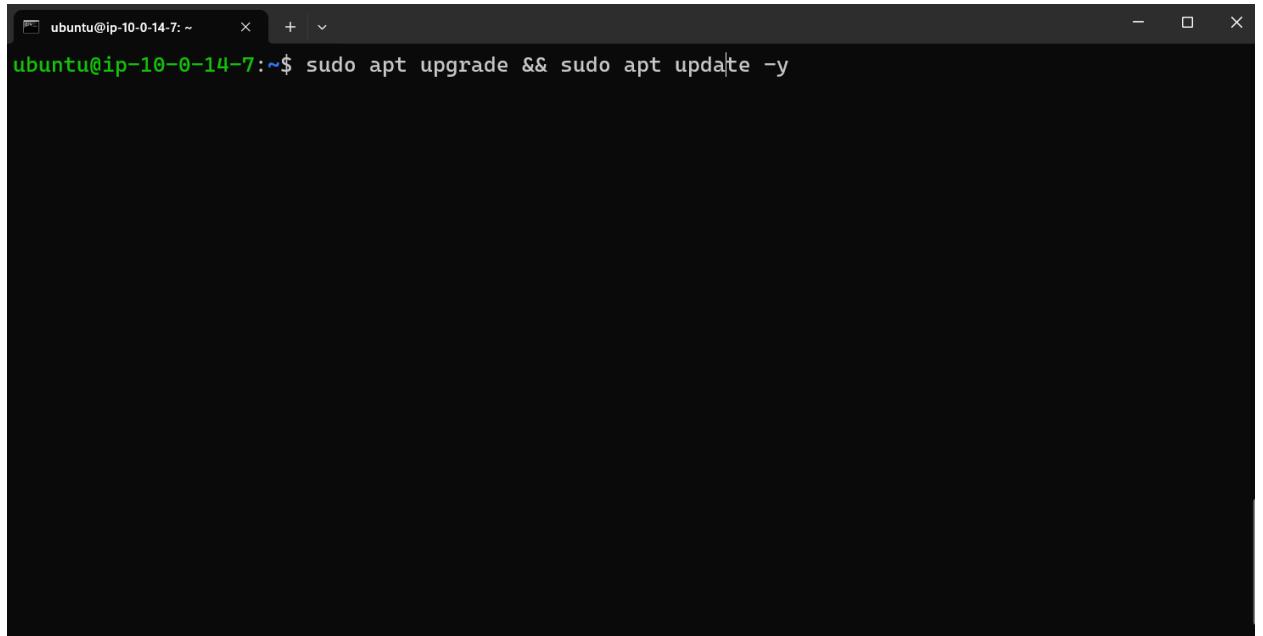
Step 3: Configuring Auto Scaling Group (ASG)

To ensure high availability and scalability of our Django application, we set up an **Auto Scaling Group (ASG)**. This setup automatically adjusts the number of EC2 instances based on demand, maintaining performance while optimizing costs.

Step 3.1 Application Setup on EC2 Instance

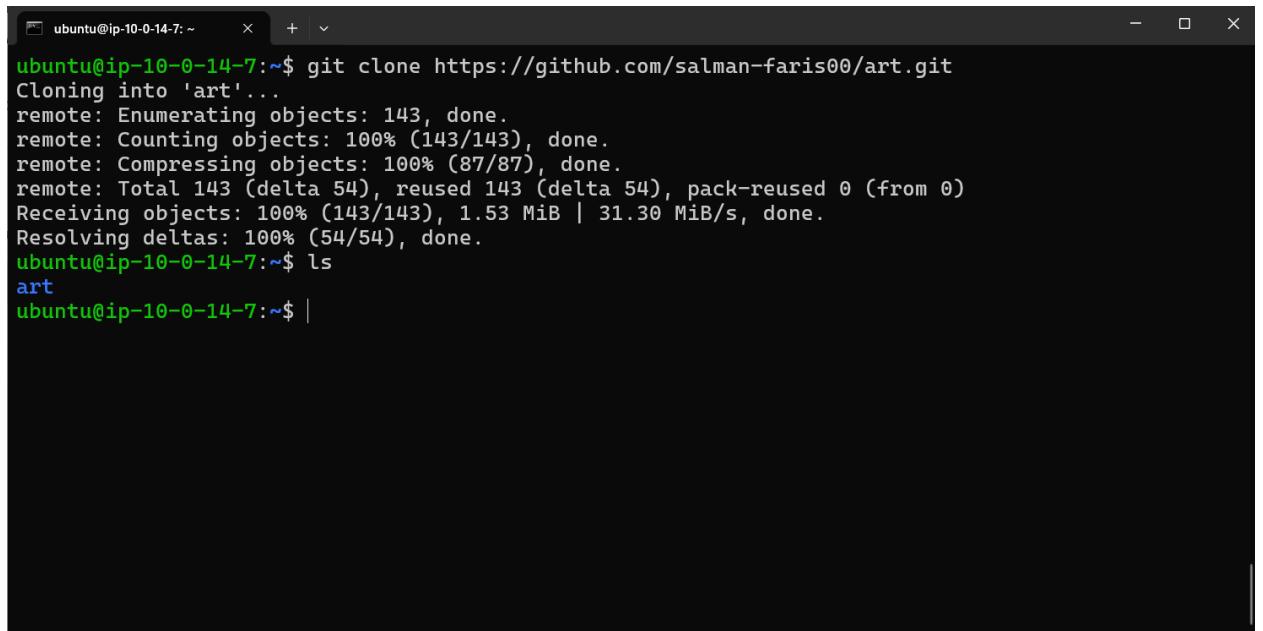
We begin by **launching a temporary EC2 instance** with ubuntu AMI. On this instance, we perform the following tasks:

- First update the ubuntu server



A screenshot of a terminal window titled "ubuntu@ip-10-0-14-7: ~". The window has a dark theme with a horizontal bar at the top featuring colored segments (brown, teal, orange, brown). The terminal displays the command "sudo apt upgrade && sudo apt update -y" in green text.

- Clone the Django project from GitHub.



A screenshot of a terminal window titled "ubuntu@ip-10-0-14-7: ~". The window has a dark theme with a horizontal bar at the top featuring colored segments (brown, teal, orange, brown). The terminal displays the command "git clone https://github.com/salman-faris00/art.git" followed by the output of the cloning process, including object enumeration, counting, compressing, and receiving objects, and finally resolving deltas. It ends with the command "ls" and the directory name "art".

- After that install python and other dependencies and nginx server the command is

```
sudo apt install python3-pip python3-dev nginx default-libmysqlclient-dev
pkg-config build-essential mysql-server -y
```

```
ubuntu@ip-10-0-14-7:~/art$ sudo apt install python3-pip python3-dev nginx python3-venv default-libmysqlclient-dev build-essential -y|
```

This will install Python, pip, and nginx server.

- To connect the RDS MySQL database from the EC2 instance, ensure that the EC2 and RDS instances are within the same VPC and have appropriate security group settings. The security group attached to the RDS should allow inbound traffic on port **3306** from the EC2 instance's security group. Once the connectivity is ensured, connect to your EC2 instance via SSH. From the EC2 terminal, you can use the MySQL client to connect to the RDS instance using the following command:

```
mysql -h <rds-endpoint> -u <username> -p
```

```
ubuntu@ip-10-0-14-7:~/art$ sudo mysql -u admin -h database-1.c0rgiwca00s9.us-east-1.rds.amazonaws.com -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 50
Server version: 8.0.41 Source distribution

Copyright (c) 2000, 2025, 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> |
```

After successfully verifying the connection to the RDS MySQL database from the EC2 instance, we have to create a database in the RDS for our application

Using sql command:

Create database art;

(art is the database name)

```
type help; or \h for help. Type \c
mysql> create database art;
Query OK, 1 row affected (0.02 sec)

mysql> |
```

After that exit from the database by performing ***Exit*** command.

- Now we have to create and activate python virtual environment for that run the below command
- To install virtual environment :

sudo apt install python3-venv -y

- Then activate python venv:

```
ubuntu@ip-10-0-14-7:~/art$ python3 -m venv env|
```

Now activate it

```
ubuntu@ip-10-0-14-7:~/art$ source env/bin/activate
(env) ubuntu@ip-10-0-14-7:~/art$ |
```

Now install all the dependencies

```
(env) ubuntu@ip-10-0-14-7:~/art$ pip install -r requirement.txt
```

After that Connecting the Django Application to the RDS Database

After applying the migrations, the next step is to connect our Django application to the MySQL database hosted in Amazon RDS.

To do this, we need to update the DATABASES configuration inside the `settings.py` file of our Django project.

In the `settings.py` file, locate the DATABASES section and replace the default database settings with the following:

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.mysql',  
        'NAME': 'art',  
        'USER': 'admin',  
        'PASSWORD': '123456789',  
        'HOST': 'database-1.c0rgiwca00s9.us-east-1.rds.amazonaws.com',  
        'PORT': '3306',  
    }  
}
```

The path of this file is `~/art/art/settings.py`

There you want to give your database ie, RDS endpoint,password,username.

Then perform the command given below:

```
(env) ubuntu@ip-10-0-14-7:~/art$ python3 manage.py makemigrations|
```

Since our primary focus in this project is on the infrastructure setup rather than the application development, we don't need to dive deeply into the details of this command. However, it's still necessary to run the following command to prepare the Django application's database migrations.

This command detects changes made to the models in the Django application and prepares migration files, which are required for setting up the database schema.

After generating the migration files using the `makemigrations` command, we need to apply those migrations to the database. This will create the required tables and schema in the RDS MySQL instance.

Run the following command:

```
(env) ubuntu@ip-10-0-14-7:~/art$ python3 manage.py migrate
```

This command executes the SQL statements defined in the migration files and sets up the database schema needed by the Django application.

Then also perform this command :

```
(env) ubuntu@ip-10-0-18-241:~/art$ python3 manage.py collectstatic
176 static files copied to '/home/ubuntu/art/staticfiles'.
(env) ubuntu@ip-10-0-18-241:~/art$ ls
```

Now Install the **gunicorn**

Gunicorn (Green Unicorn) is a **Python WSGI HTTP server** used to serve Python web applications like Django in a production environment. It acts as a bridge between your Django application and web servers like NGINX.

Gunicorn is:

- **Lightweight** and **easy to configure**.
- Supports **multiple workers**, allowing better handling of concurrent requests.
- Designed to work behind a **reverse proxy server** like NGINX for improved performance and security.

Gunicorn does **not serve static files**, which is why we use NGINX alongside it in production setups.

- **Installation Steps**

SSH into your EC2 instance and activate your Python virtual environment (if used). Then run:

```
(env) ubuntu@ip-10-0-14-7:~/art$ pip install gunicorn django
```

Now we installed gunicorn and django.

To confirm it was installed correctly:

```
(env) ubuntu@ip-10-0-14-7:~/art$ gunicorn --version
gunicorn (version 23.0.0)
(env) ubuntu@ip-10-0-14-7:~/art$ |
```

- **Test Gunicorn with Django**

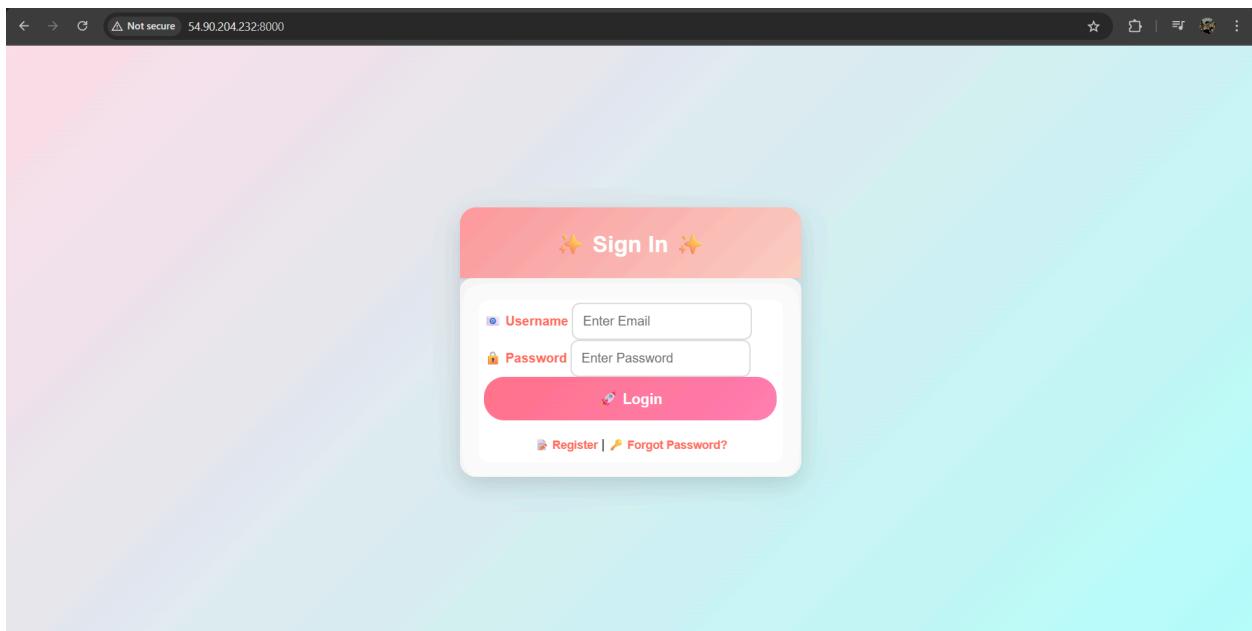
Navigate to your Django project directory (where `manage.py` is located), and run:

Gunicorn -bind 0.0.0.0:8000 art.wsgi:application

```
(env) ubuntu@ip-10-0-14-7:~/art$ gunicorn --bind 0.0.0.0:8000 art.wsgi:application
[2025-07-26 15:17:20 +0000] [3012] [INFO] Starting gunicorn 23.0.0
[2025-07-26 15:17:20 +0000] [3012] [INFO] Listening at: http://0.0.0.0:8000 (3012)
[2025-07-26 15:17:20 +0000] [3012] [INFO] Using worker: sync
[2025-07-26 15:17:20 +0000] [3013] [INFO] Booting worker with pid: 3013
```

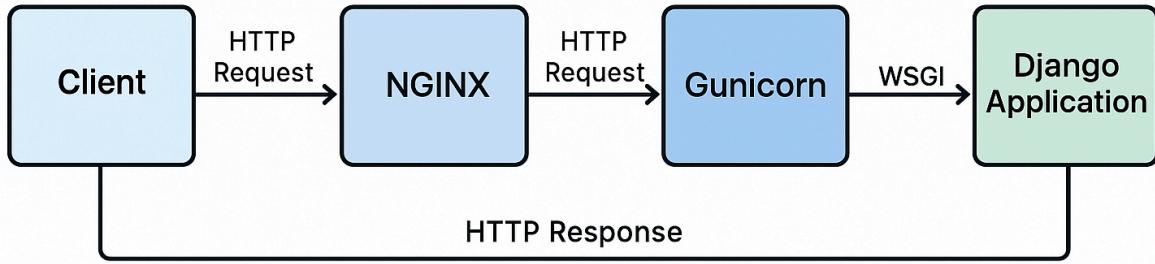
Now access the website with your public ip along with the 8000 port number ,also update the security group of your instance open 8000 port.

You will be getting output like this:



Now if you access the website you can't see the static files like images, this is because the unicorn server is can't serve the static files .

For that we will be using nginx also we will configure nginx as a proxy server



Nginx is a powerful **reverse proxy** that handles HTTP requests and forwards them to Gunicorn. It also serves static files (CSS, JS, images) directly to clients, which offloads that responsibility from your Django app.

To connect nginx and gunicorn here we are using unix socket file:

A **Unix domain socket** is a special file used for **inter-process communication (IPC)** on the same server. Instead of using a network port (like 127.0.0.1:8000), Gunicorn and Nginx communicate via a file path, like /home/ubuntu/project/art.sock.

Benefits of Using Unix Socket with Gunicorn and Nginx

1. Faster than TCP/IP

- 
- **No network overhead:** Communication happens within the operating system kernel, without going through the network stack.
 - Result: **Lower latency** and **faster response times** for local app-server communication.

2. More Secure

- Unlike a TCP port that can potentially be exposed to the network, a socket file **only exists on the local filesystem**.
- Only users with access to the file can use it, making it harder to attack.

3. Simpler for Local Communication

- Since both Nginx and Gunicorn run on the same machine, Unix sockets are **a perfect fit** for local-only communication.
- It avoids unnecessary complexity related to networking (IP/Port management).

4. Less Resource Consumption

- Uses **fewer system resources** than opening a TCP port.
- Helps in high-performance environments with minimal system load.

5. Easy Integration with Systemd

- Systemd service files can manage socket files cleanly.
- You can include permissions, ownership, and automatic recreation during restarts.

When Not to Use Socket Files

- If your Gunicorn server and Nginx server run on **separate machines**.

- 
- If you need **load balancing** across multiple Gunicorn servers (use TCP/IP for that).

To **create and use a Gunicorn socket file** for your Django project, follow these detailed steps. This setup is crucial for integrating Gunicorn with Nginx as a **reverse proxy** using a **Unix socket**.

Create a systemd service file to manage Gunicorn with a socket.

```
sudo nano /etc/systemd/system/gunicorn.service
```

Paste the following configuration

[Unit]

Description=Gunicorn daemon for Django project

After=network.target

[Service]

User=ubuntu

Group=www-data

WorkingDirectory=/home/ubuntu/yourproject

*ExecStart=/home/ubuntu/yourproject/env/bin/gunicorn *

*--workers 3 *

```
--bind unix:/home/ubuntu/yourproject/yourproject.sock \
yourproject.wsgi:application
```

[Install]

WantedBy=multi-user.target

```
GNU nano 7.2                               /etc/systemd/system/gunicorn.service
[Unit]
Description=Gunicorn daemon for Django project
After=network.target

[Service]
User=ubuntu
Group=www-data
WorkingDirectory=/home/ubuntu/art
ExecStart=/home/ubuntu/art/env/bin/gunicorn \
          --workers 3 \
          --bind unix:/home/ubuntu/art/art.sock \
          art.wsgi:application

[Install]
WantedBy=multi-user.target

[Read 15 lines]
```

- **WorkingDirectory:** Directory where your `manage.py` is located.
- **ExecStart:** Starts Gunicorn with 3 workers, binding it to a **Unix socket file** instead of a port.
- **--bind unix:/path/to/project.sock:** This creates a socket file that Nginx will use to communicate with Gunicorn.

Reload and Start the Gunicorn Service

sudo systemctl daemon-reload

sudo systemctl start gunicorn

sudo systemctl enable gunicorn

To verify that the socket file is created and Gunicorn is running:

sudo systemctl status gunicorn

```
(env) ubuntu@ip-10-0-14-7:~/art$ sudo systemctl status gunicorn
● gunicorn.service - Gunicorn daemon for Django project
   Loaded: loaded (/etc/systemd/system/gunicorn.service; enabled; preset: enabled)
   Active: active (running) since Sat 2025-07-26 15:52:35 UTC; 2min 54s ago
     Main PID: 3507 (gunicorn)
        Tasks: 4 (limit: 1072)
       Memory: 97.3M (peak: 98.3M)
          CPU: 974ms
        CGroup: /system.slice/gunicorn.service
                ├─3507 /home/ubuntu/art/env/bin/python3 /home/ubuntu/art/env/bin/gunicorn --workers 3 --bind unix:/home/ubuntu/art.sock
                ├─3554 /home/ubuntu/art/env/bin/python3 /home/ubuntu/art/env/bin/gunicorn --workers 3 --bind unix:/home/ubuntu/art.sock
                ├─3555 /home/ubuntu/art/env/bin/python3 /home/ubuntu/art/env/bin/gunicorn --workers 3 --bind unix:/home/ubuntu/art.sock
                ├─3556 /home/ubuntu/art/env/bin/python3 /home/ubuntu/art/env/bin/gunicorn --workers 3 --bind unix:/home/ubuntu/art.sock

Jul 26 15:52:35 ip-10-0-14-7 systemd[1]: Started gunicorn.service - Gunicorn daemon for Django project.
Jul 26 15:52:35 ip-10-0-14-7 gunicorn[3507]: [2025-07-26 15:52:35 +0000] [3507] [INFO] Starting gunicorn 23.0.0
Jul 26 15:52:35 ip-10-0-14-7 gunicorn[3507]: [2025-07-26 15:52:35 +0000] [3507] [INFO] Listening at: unix:/home/ubuntu/art.sock
Jul 26 15:52:35 ip-10-0-14-7 gunicorn[3507]: [2025-07-26 15:52:35 +0000] [3507] [INFO] Using worker: sync
Jul 26 15:52:35 ip-10-0-14-7 gunicorn[3554]: [2025-07-26 15:52:35 +0000] [3554] [INFO] Booting worker with pid: 3554
Jul 26 15:52:35 ip-10-0-14-7 gunicorn[3555]: [2025-07-26 15:52:35 +0000] [3555] [INFO] Booting worker with pid: 3555
Jul 26 15:52:35 ip-10-0-14-7 gunicorn[3556]: [2025-07-26 15:52:35 +0000] [3556] [INFO] Booting worker with pid: 3556
lines 1-20/20 (END)
```

Configure NGINX to Use the Socket File

Edit your NGINX config

sudo nano /etc/nginx/sites-available/art

Example configuration:

```
GNU nano 7.2                               /etc/nginx/sites-available/art
server {
    listen 80;
    server_name 54.90.204.232;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        alias /home/ubuntu/art/staticfiles/;
    }

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/ubuntu/art/art.sock;
    }
}
```

Then enable the site and reload Nginx:

sudo ln -s /etc/nginx/sites-available/art /etc/nginx/sites-enabled

sudo nginx -t

sudo systemctl restart nginx

Now if you access your public ip you will be getting your website ,if your are getting error like 502 bad gateway



Give Permissions (If You Face 502 or Forbidden Errors)

Make sure the ubuntu user and www-data group have access to the socket file:

sudo chown ubuntu:www-data /home/ubuntu/art/art.sock

sudo chmod 770 /home/ubuntu/art/art.sock

Also ensure the parent directory is executable:

sudo chmod o+x /home

```
sudo chmod o+x /home/ubuntu
```

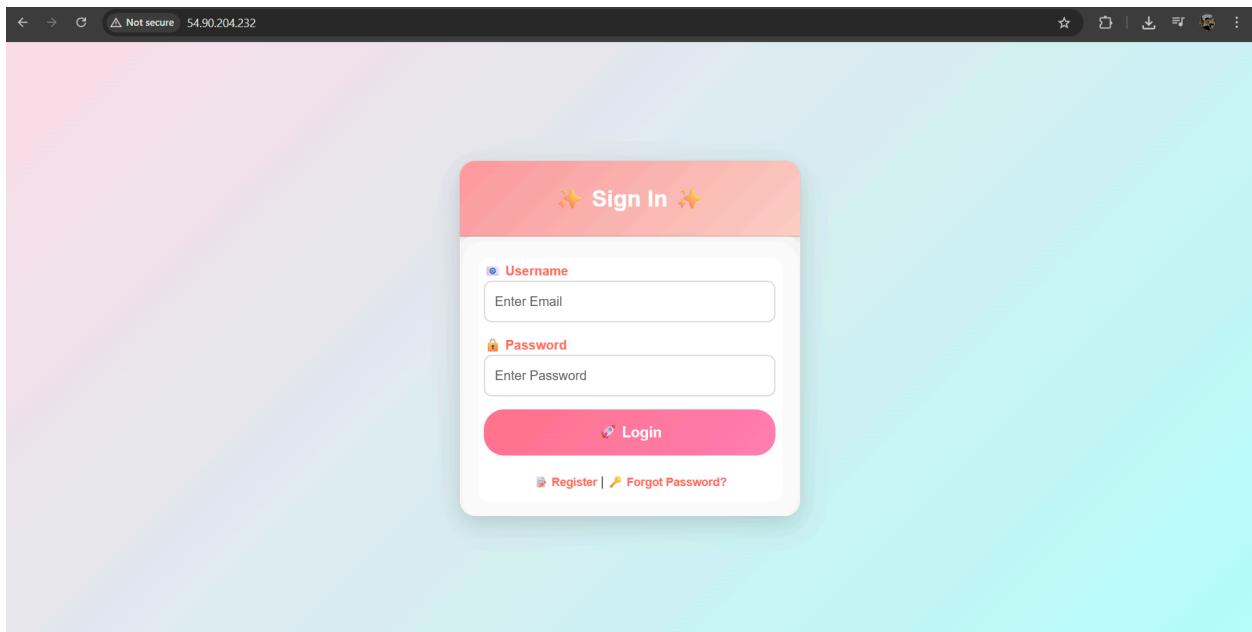
```
sudo chmod o+x /home/ubuntu/art
```

Then restart the servers:

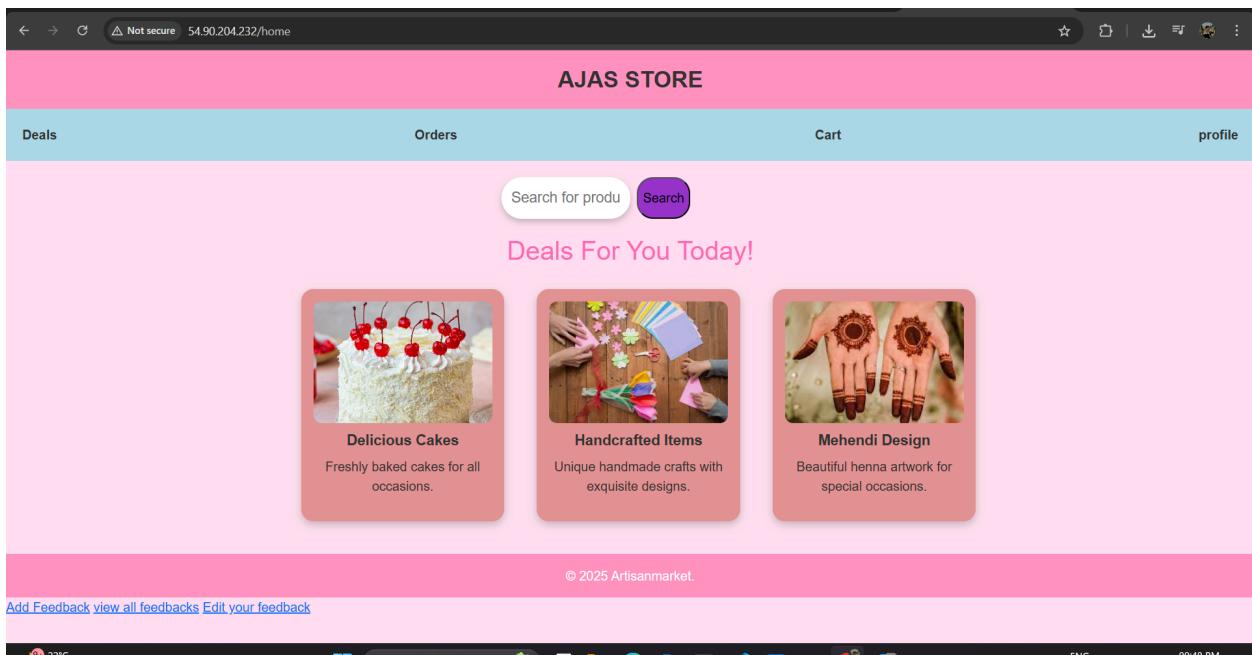
```
sudo systemctl restart gunicorn
```

```
sudo systemctl restart nginx
```

Then access the website again:



Now the nginx is also serving the staticfiles like images



Now Access the Django application using a **custom domain name** instead of the EC2 **public IP address**, by bringing DNS management from **GoDaddy** to **AWS Route 53**.

For that make some changes in nginx configuration file (/etc/nginx/sites-available/art):

```
server {
    listen 80;
    server_name 54.90.204.232;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        alias /home/ubuntu/art/staticfiles/;
    }
}
```

Edit server_name with your custom domain:

```

server {
    listen 80;
    server_name mysample.xyz www.mysample.xyz;

    location = /favicon.ico { access_log off; log_not_found off; }
    location /static/ {
        alias /home/ubuntu/art/staticfiles/;
    }
}

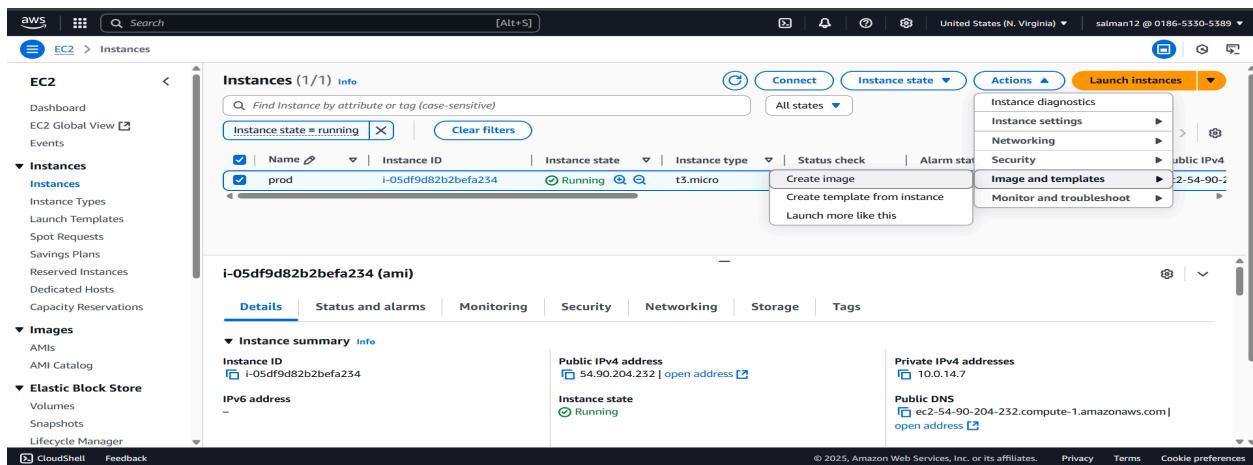
```

After updating the Django settings, restart and enable both **Gunicorn** and **Nginx** to apply the changes and ensure the services start automatically on boot.

To ensure high scalability and availability, we will use AWS Auto Scaling. For that, we must first **create a Launch Template**. This template will be based on the **Amazon Machine Image (AMI)** of the current EC2 instance where our Django application is configured and running successfully.

Steps:

1. Create an AMI from the current EC2 instance.
2. Use this AMI while creating the Launch Template.

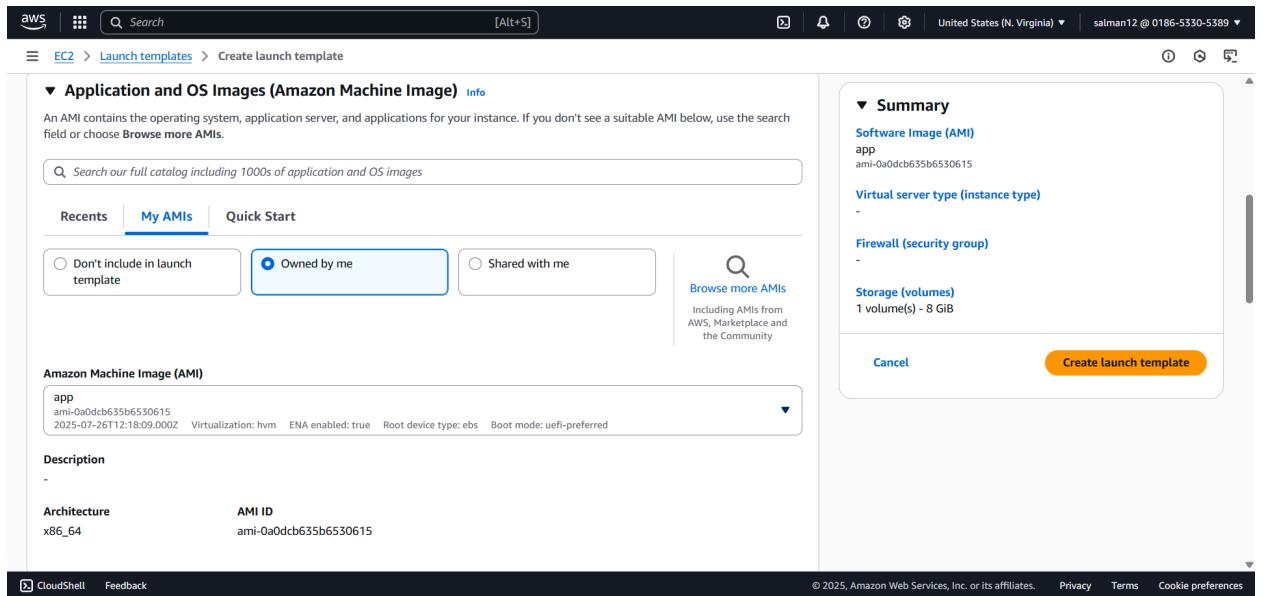


Step 3.2: Create Launch Template with IAM Role Access to S3

After creating an AMI (Amazon Machine Image) from the EC2 instance where our Django application is configured and tested, the next step is to create a **Launch Template**. This template will be used by the **Auto Scaling Group** to launch new instances automatically when needed.

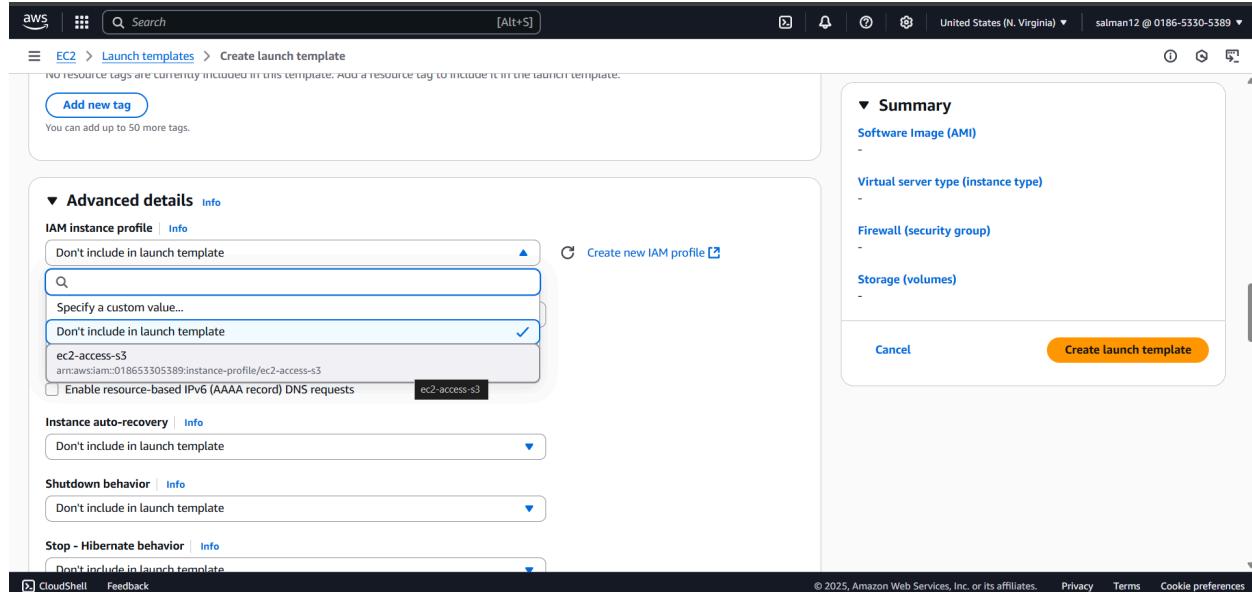
While creating the launch template, we perform the following:

- **Select the AMI:** Use the previously created AMI that contains the pre-configured Django application, Gunicorn server, and required dependencies.



- **Assign an IAM Role:** Attach an IAM role to the launch template that grants the EC2 instances permission to **write application logs to Amazon S3**. This is done securely

via a **VPC Gateway Endpoint**, ensuring the traffic does not traverse the public internet.



- **Configure Security Groups:** Apply the appropriate security group to allow communication with the Application Load Balancer and other required services.

This launch template ensures that every instance created by the Auto Scaling Group will be identical, secure, and production-ready with S3 access configured from the start.

Step 3.3: Create Auto Scaling Group and Attach Application Load Balancer

After setting up the **Launch Template** with the necessary AMI and IAM role, the next step is to create the **Auto Scaling Group (ASG)**. Auto Scaling ensures that your application is highly available and fault-tolerant by automatically launching or terminating EC2 instances based on traffic demand.

Steps to Create the Auto Scaling Group:

1. Choose the Launch Template

- Start by selecting the previously created launch template.
- This template already includes the AMI with the Django app and logging configurations, and the IAM role to access S3.

2. Configure the VPC and Subnets

- Select the VPC created for this project.
- Choose **private subnets** for instance deployment to enhance security, as application servers should not be directly accessible from the internet.

3. Attach an Application Load Balancer (ALB)

- Select an existing **Application Load Balancer** (or create one if not already done).
- The ALB will be placed in the **public subnets** to accept incoming traffic.
- Register the Auto Scaling Group with the **target group** associated with the ALB, ensuring that requests are forwarded to healthy EC2 instances in the private subnets.

Security and Availability Considerations:

- Traffic enters via the ALB in public subnets and is routed to the EC2 instances in private subnets.
- Only the ALB can access these instances, ensuring strong **network isolation and security**.
- The IAM role attached to the launch template ensures that instances can log application data to **S3 through VPC endpoints**.

This setup ensures your Django application is **scalable, highly available**, and **secure**, ready to handle production workloads efficiently.

Step4: Using S3 Gateway VPC Endpoint for Secure Log Delivery

To ensure that ALB access logs are delivered to S3 **without traversing the public internet**, we use an **S3 Gateway VPC Endpoint**.

- Create a s3 bucket for collect access logs
- Goto loadbalancer → attributes→

- There turn on the access logs and select the s3 bucket

You can also give the prefix for storing access logs and connection logs

Step5 : Migrating Domain DNS from GoDaddy to Route 53

Access the Django application using a **custom domain name** instead of the EC2 **public IP address**, by bringing DNS management from **GoDaddy** to **AWS Route 53**.

Create a Hosted Zone in Route 53

1. Go to the Route 53 console.
2. Click "Hosted Zones" → "Create hosted zone"

3. Enter:

- **Domain Name:** mysample.xyz (your domain)
- **Type:** Public hosted zone

Create hosted zone [Info](#)

Hosted zone configuration

A hosted zone is a container that holds information about how you want to route traffic for a domain, such as example.com, and its subdomains.

Domain name [Info](#)
This is the name of the domain that you want to route traffic for.

Valid characters: a-z, 0-9, !* # \$ % & ` () * +,- / ; <= > ? @ [\] ^ _ { } { } , ~

Description - optional [Info](#)
This value lets you distinguish hosted zones that have the same name.

The description can have up to 256 characters. 0/256

Type [Info](#)
The type indicates whether you want to route traffic on the internet or in an Amazon VPC.

Public hosted zone
A public hosted zone determines how traffic is routed on the internet.

Private hosted zone
A private hosted zone determines how traffic is routed within an Amazon VPC.

4. Click **Create**

Copy the AWS Name Servers

Once the hosted zone is created, you'll see 4 NS (Name Server) records like:

mysample.xyz [Info](#)

Hosted zone details

[Edit hosted zone](#)

Records (2) [DNSSEC signing](#) [Hosted zone tags \(0\)](#)

Records (2) [Info](#)

Automatic mode is the current search behavior optimized for best filter results. [To change modes go to settings.](#)

	Record ...	Type	Routin...	Differ...	Alias	Value/Route traffic to	TTL (s...)	Health ...	Evalu...
<input type="checkbox"/>	mysample...	NS	Simple	-	No	ns-1230.awsdns-25.org. ns-1984.awsdns-56.co.uk. ns-413.awsdns-51.com. ns-894.awsdns-47.net.	172800	-	-
<input type="checkbox"/>	mysample...	SOA	Simple	-	No	ns-1230.awsdns-25.org. aws...	900	-	-

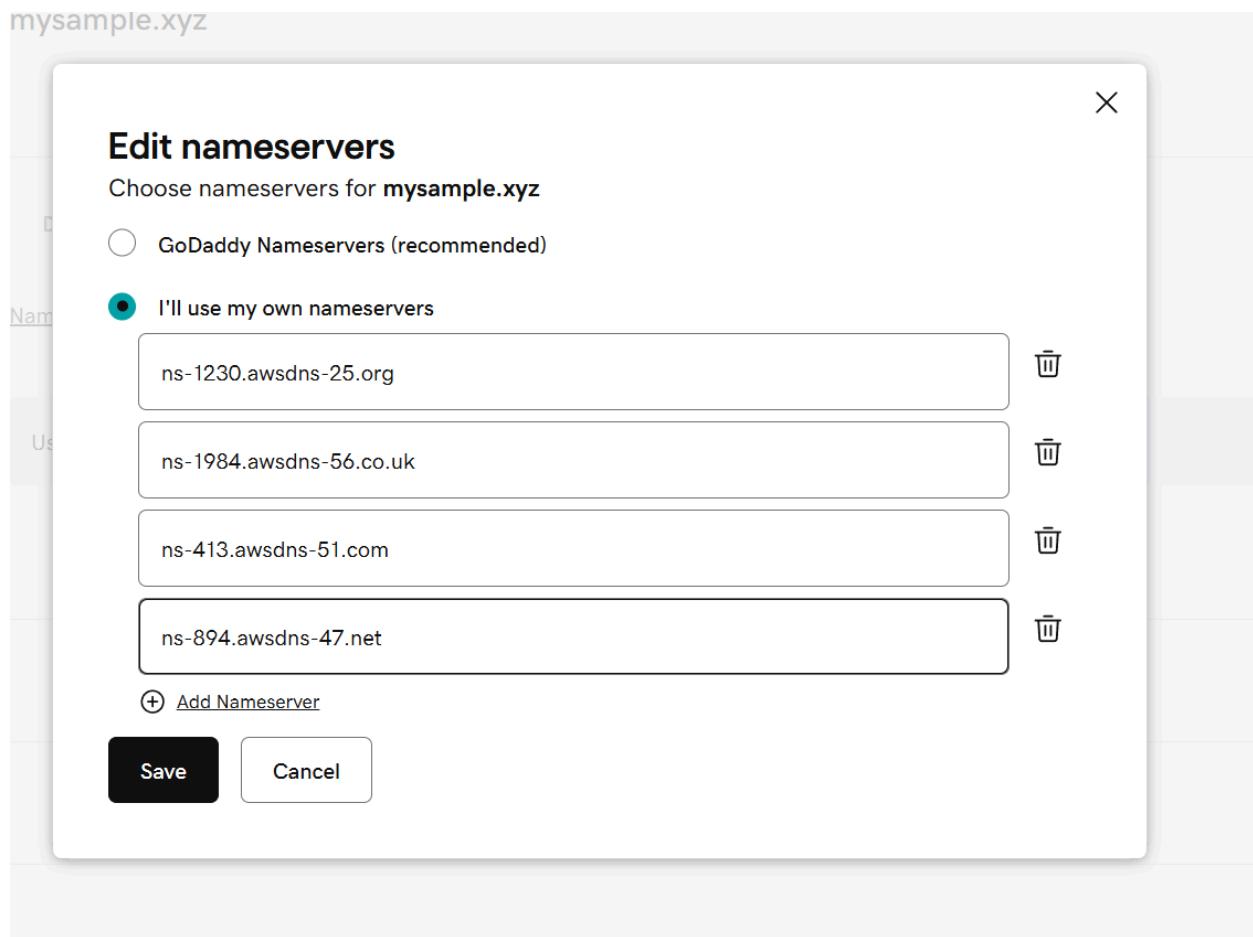
These are the **AWS Route 53 name servers** you will configure in GoDaddy.

Update Name Servers in GoDaddy

1. Log in to your GoDaddy Domain Manager.
2. Click on your domain (`mysample.xyz`)
3. Go to **DNS > Nameservers**

The screenshot shows the GoDaddy Domain Manager interface for the domain `mysample.xyz`. The top navigation bar includes a **Connect Domain** button. Below the domain name, there are tabs for **Overview**, **DNS** (which is selected and highlighted in grey), **Registration Settings**, and **Products**. Under the **DNS** tab, there are sub-tabs for **DNS Records**, **Forwarding**, **Nameservers** (which is selected and highlighted in grey), and **Hostnames**. A note below the sub-tabs states: Nameservers determine where your DNS is hosted and where you add, edit or delete your DNS records. A message at the top of the main content area says "Using default nameservers". On the right side of this message, there is a **Change Nameservers** button. Below this, the section title **Nameservers** is followed by a help icon (a question mark in a circle). Two nameserver entries are listed: `ns21.domaincontrol.com` and `ns22.domaincontrol.com`.

4. Click **Change**
5. Choose **Enter my own nameservers (advanced)**
6. Paste the **4 name servers** copied from Route 53



7. Click **Save**

Now your domain DNS is managed by AWS Route 53.

Step6 : Issuing an ACM Certificate for HTTPS (SSL)

To secure your website with **HTTPS**, you need an **SSL certificate**. AWS provides free certificates through **ACM (AWS Certificate Manager)**. These certificates are required to enable HTTPS for your domain on **CloudFront**

Step 1: Open ACM Console

1. Go to AWS Certificate Manager (ACM).

2. Make sure your region is N. Virginia (us-east-1) – required for CloudFront.

Step 2: Request a Certificate

1. Click “Request a certificate”
2. Choose Public Certificate
3. Click Next

Step 3: Add Domain Names

- Enter your domain name:
mysample.xyz

The screenshot shows the 'Request public certificate' step. In the 'Domain names' section, a 'Fully qualified domain name' input field contains 'mysample.xyz'. Below it is a button labeled 'Add another name to this certificate'. At the bottom, there is a link 'Allow export'.

4. Click Next

Step 4: Select Validation Method

- Choose DNS validation (easier if you're using Route 53)

The screenshot shows the 'Validation method' selection step. It offers two choices: 'DNS validation - recommended' (which is selected with a blue radio button) and 'Email validation' (which is unselected with a grey radio button). Descriptions for each option are provided below them.

5. Click Next and then Request

Step 5: Add DNS Validation Record

- After submitting, you'll see a CNAME record that must be added to your DNS settings.

Domains (1)				
Domain	Status	Renewal status	Type	CNAME name
mysample.xyz	Pending validation	-	CNAME	_9dbb9791e1c2d981abadfdb59db75d83.mysam

- If your domain is managed by Route 53:
 - Click "Create record in Route 53" → ACM will add the validation record automatically.

Create DNS records in Amazon Route 53 (1/1)

Q Search domains 1 match

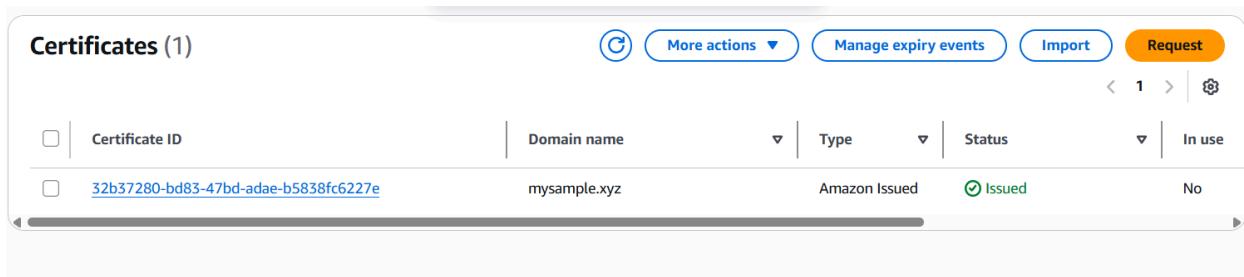
Validation status = Pending validation X Validation status = Failed X Is domain in Route 53? = Yes X Clear filters < 1 >

Domain	Validation status	Is domain in Route 53?
mysample.xyz	Pending validation	Yes

Cancel Create records

Step 6: Wait for Validation

- ACM will issue the certificate once the DNS record is verified (usually within a few minutes).
- The Status will change from "Pending validation" to "Issued"



The screenshot shows the ACM Certificates page with one certificate listed:

Certificate ID	Domain name	Type	Status	In use
32b37280-bd83-47bd-adae-b5838fc6227e	mysample.xyz	Amazon Issued	Issued	No

- **HTTPS Listener (Add Port 443) to ALB Using ACM Certificate**

To allow secure **HTTPS** traffic on port **443** through your **Application Load Balancer (ALB)** using an **SSL certificate** from **AWS Certificate Manager (ACM)**.

Go to EC2 Console → Load Balancers

Navigate to **EC2 → Load Balancers**, and select your ALB.

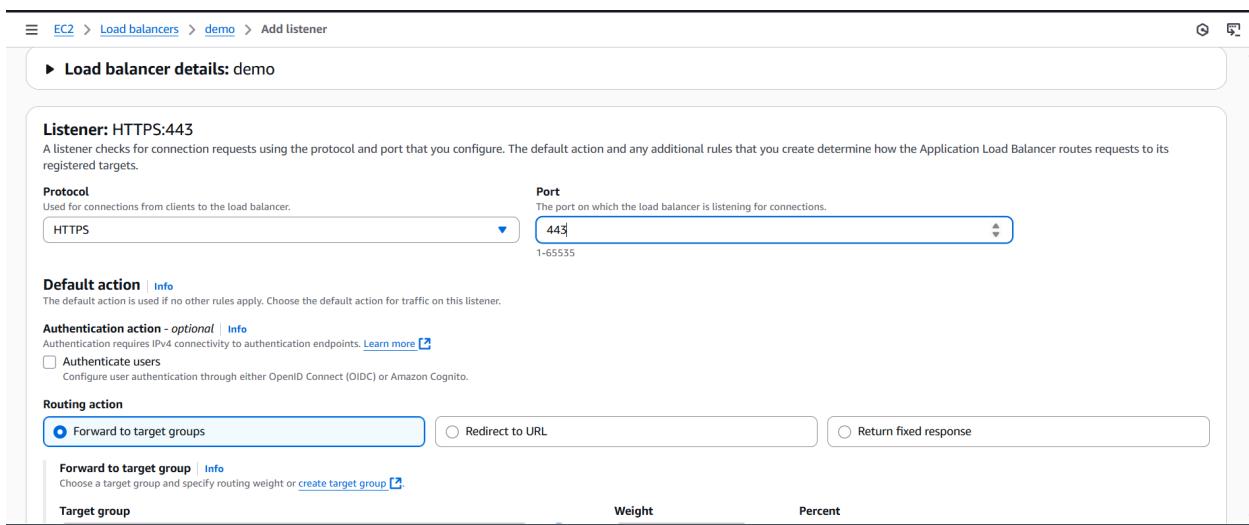
Click on “Listeners” tab

You'll see existing listeners (like HTTP - port 80).

Click “Add Listener”

- **Protocol: HTTPS**

- **Port: 443**



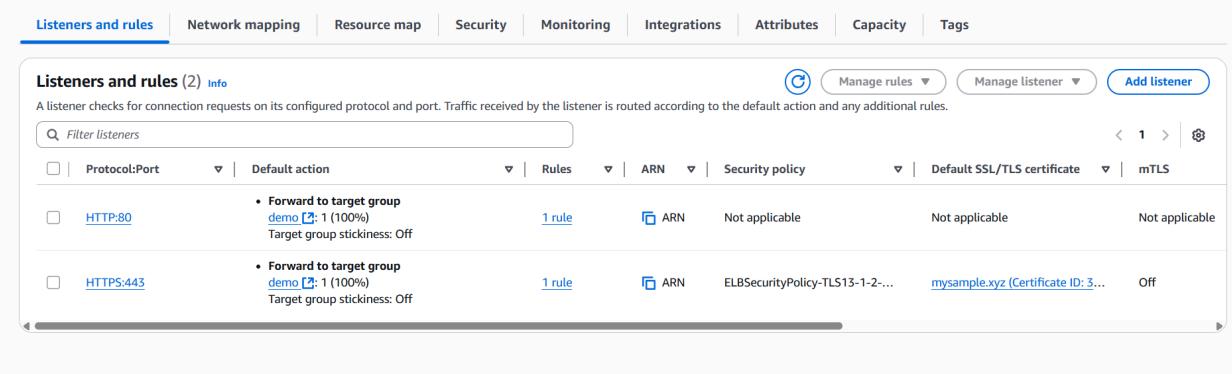
The screenshot shows the 'Load balancer details: demo' configuration page for an Application Load Balancer (ALB). The 'Listener: HTTPS:443' section is active, showing the following settings:

- Protocol:** HTTPS (selected)
- Port:** 443
- Default action:** Forward to target groups (selected)
- Authentication action - optional:** None selected
- Routing action:** Forward to target groups (selected)

Below the configuration, there is a table for defining target groups:

Target group	Weight	Percent

- Now there will be having two listeners



The screenshot shows the AWS CloudWatch Metrics Insights interface. A query is being run against the CloudFront Metrics dataset. The query is:

```
CloudFront Metrics | WHERE CloudFront Metrics.Timestamp >= ago(1h) | GROUP BY CloudFront Metrics.CloudFront Distribution ARN, CloudFront Metrics.CloudFront Distribution Origin, CloudFront Metrics.CloudFront Distribution Protocol, CloudFront Metrics.CloudFront Distribution Listener Port | METRICS CloudFront Metrics.CloudFront Distribution Requests.Count, CloudFront Metrics.CloudFront Distribution Requests.Sum | Dims CloudFront Metrics.CloudFront Distribution Origin, CloudFront Metrics.CloudFront Distribution Listener Port
```

The results show two listeners: **HTTP:80** and **HTTPS:443**. The **HTTP:80** listener has one rule that forwards traffic to a target group named **demo** (100% weight). The **HTTPS:443** listener also has one rule that forwards traffic to the same target group **demo** (100% weight). Both listeners have their target group stickiness set to Off.

Step 7 : Setting Up CloudFront in Front of the Load Balancer

Place Amazon CloudFront, a global content delivery network (CDN), in front of your Application Load Balancer (ALB) to:

- Improve latency and performance globally
- Add an additional layer of DDoS protection (via AWS Shield)
- Enable HTTPS using Amazon Certificate Manager (ACM)
- Cache static/dynamic content efficiently

Step 1: Create a CloudFront Distribution

- Go to the CloudFront Console.
- Click Create Distribution
- Under Origin Settings:
 - Origin domain: Paste your Load Balancer DNS name (e.g., `myapp-alb-1234567890.us-east-1.elb.amazonaws.com`)
- You can also integrate with WAF for additional security

- You can also give your custom domain while creating the distribution or you can do it later

Custom domain - optional Info

Domain

Use your own custom domain with free HTTPS to provide a secure, friendly URL for your app. You can add a custom domain later if you do not have a Route 53 zone in this account.

Check domain

- Here we are giving the domain after the cloud front distribution
- Then create the distribution

ID	Status	Description	Type	Domain name (s...)	Alternate domai...	Origins	Last modified
E2M9KDEPAZLOLZ	Enabled	-	Standard	dccy8dz96p7q...	-	demo-1504030737.us-e...	July 28, 2025, 09:2...

- Now get inside the distribution there you can specify your domain name by editing the alternative domain name

Settings

Description
-

Price class
Use all edge locations (best performance)

Supported HTTP versions
HTTP/2, HTTP/1.1, HTTP/1.0

Alternate domain names

Add domain

Standard logging
[Off](#)

Cookie logging
[Off](#)

Default root object
-

Edit

- Add domain

Configure domains

Choose the domains that will be served by this distribution.

Domains

Domains to serve

CloudFront will serve the following domains. They must all be under the same root domain.

- mysample.xyz

[Remove](#)

[Add another domain](#)

[Cancel](#) [Next](#)

- Here we have to select the TLS/SSL certificate that we create using the ACM

Get TLS certificate

TLS certificate [Info](#) [Refresh certificates](#)

Transport layer security (TLS) encrypts communication to and from your domain. You must have a TLS certificate with AWS Certificate Manager (ACM) to use CloudFront.

Available certificates [Info](#)

These certificates cover the domains that will be served by this distribution.

- mysample.xyz (32b37280-bd83-47bd-adae-b5838fc6227e)
- Create a new certificate

Certificate details

ARN arn:aws:acm:us-east-1:018653305389:certificate/32b37280-bd83-47bd-adae-b5838fc6227e	Covered domains mysample.xyz	Source Amazon
---	--	-------------------------

[View in AWS Certificate Manager](#) [Edit](#)

[Cancel](#) [Previous](#) [Next](#)

- Then click add domain
- Then you can see like this

Settings

Description
-

Price class
Use all edge locations (best performance)

Supported HTTP versions
HTTP/2, HTTP/1.1, HTTP/1.0

Alternate domain names
[mysample.xyz](#) [Edit](#)

[Route domains to CloudFront](#)

Custom SSL certificate
[mysample.xyz](#) [Edit](#)

Security policy
TLSv1.2_2021

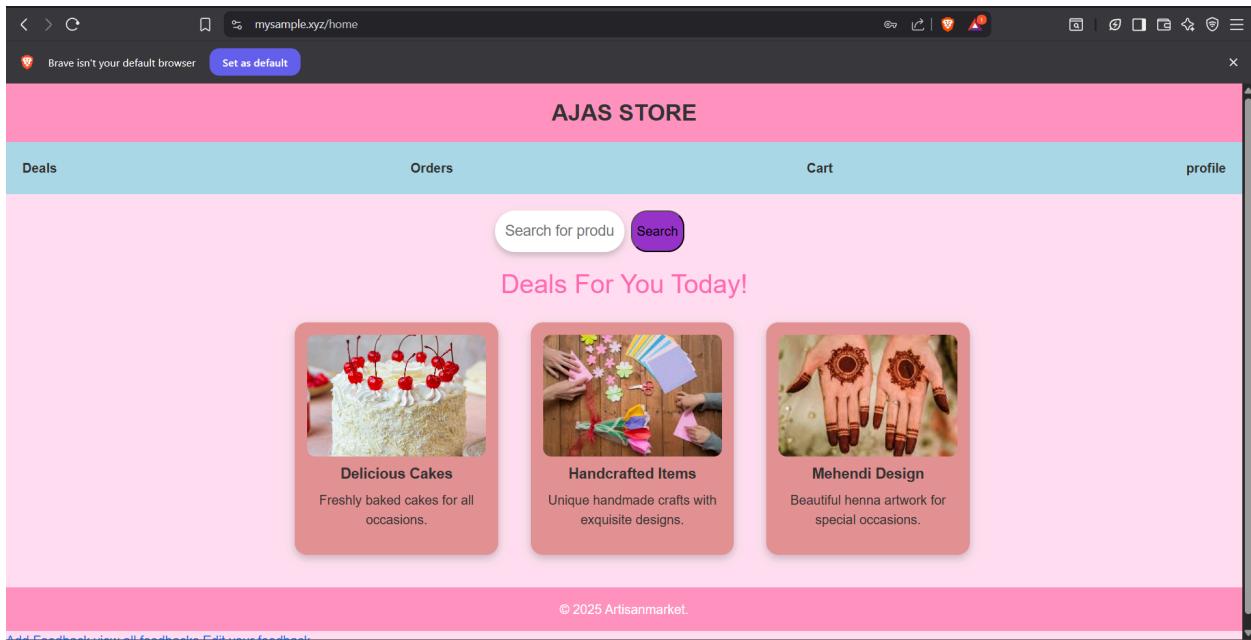
Standard logging
Off

Cookie logging
Off

Default root object
-

Continuous deployment [Info](#)

- Now if you access the domain you will be getting the website



Step8 :Monitoring with CloudWatch and SNS

In a production-grade deployment, proactive monitoring is crucial for ensuring the reliability, performance, and availability of your application. AWS provides two core services to achieve this: **Amazon CloudWatch** for monitoring metrics and logs, and **Amazon SNS (Simple Notification Service)** for sending real-time alerts.

CloudWatch Overview

Amazon CloudWatch allows you to collect, monitor, and analyze metrics across AWS services like EC2, RDS, Load Balancers, and more. With CloudWatch, you can set up custom dashboards to visualize application health, configure **alarms** to detect anomalies or failures, and dig into logs for debugging.

For example, you can create a CloudWatch alarm to monitor your **EC2 instance's CPU utilization**. If it consistently goes above 80% for a 5-minute interval, the alarm will trigger. Similarly, alarms can be created to track **memory usage, disk space, or RDS free storage space**. You can also monitor **Application Load Balancer (ALB)** health metrics to ensure traffic is properly routed to healthy targets.

Integrating SNS for Alerts

To receive alerts when an alarm is triggered, you can integrate **CloudWatch Alarms** with **Amazon SNS**. SNS is a fully managed publish/subscribe messaging service that enables you to send notifications via email, SMS, or even trigger Lambda functions.

Here's how it works:

1. You first create an **SNS topic** — a logical access point for notifications.
2. Then, subscribe your **email address or phone number** to that topic.
3. Finally, attach the SNS topic to your **CloudWatch Alarm**.

Once set up, when a defined threshold is breached (e.g., RDS storage falls below 1GB), CloudWatch sends the alarm signal to the SNS topic, which then **sends an immediate email/SMS notification** to all subscribed endpoints.

Conclusion

- This documentation has walked through the complete process of deploying a **production-grade Django application** on AWS infrastructure using best practices. We began by launching an EC2 instance and configuring it with a Django application, followed by integrating with an RDS MySQL database. We then set up **Gunicorn** as the WSGI server and used **NGINX** as a reverse proxy to efficiently serve traffic and static files. With Gunicorn sockets and systemd services, we ensured that our application starts automatically and remains stable even after reboots.
- To handle scalability and high availability, we created a **Launch Template** and set up an **Auto Scaling Group (ASG)** behind an **Application Load Balancer (ALB)**. This architecture allows automatic addition or removal of EC2 instances based on traffic, ensuring optimal resource usage and performance.
- To make the application accessible via a custom domain, we registered a domain through GoDaddy and pointed it to **Route 53** by changing the nameservers. We then issued a **free SSL certificate via AWS ACM** and routed traffic securely over

HTTPS by configuring a **CloudFront Distribution** in front of the ALB, providing better performance and security.

- In case of any issues, it's essential to check the **Security Groups** of your EC2, RDS, Load Balancer, and VPC endpoints. Ensure proper **inbound/outbound rules** are configured to allow the necessary traffic (such as ports 80, 443, and 3306).
- To keep the infrastructure reliable and proactive, we implemented monitoring using **Amazon CloudWatch**. Metrics such as CPU usage, RDS storage space, and load balancer health are tracked in real-time. Alarms are created for threshold breaches, and **Amazon SNS** is used to send notifications via email or SMS, allowing for **instant alerting** and reducing downtime risks.
- This end-to-end setup ensures your application is secure, scalable, and production-ready with observability and automation features in place.

Tab 2

