Final Project

Muhammad Asad ur Rehman Nadeem - 29456

This project demonstrates the design and deployment of a scalable, secure, and containerized infrastructure on AWS using Terraform. The objective was to automate the provisioning of cloud resources and orchestrate the deployment of a full-stack Dockerized web application, complete with database integration, load balancing, custom domain configuration, SSL encryption, and real-time business intelligence (BI) visualization.

By leveraging Infrastructure as Code (IaC) with Terraform, modular and reusable templates were developed to launch and manage critical AWS services, including EC2 instances (via Auto Scaling Groups), RDS (PostgreSQL and MySQL in private subnets), and Application Load Balancers with HTTPS support.

The frontend and backend applications were containerized using multi-stage Docker builds and deployed across EC2 instances behind an ALB. A separate EC2 instance was provisioned for the BI tool (Metabase), which was containerized using Docker and connected securely to the PostgreSQL RDS database. The BI dashboard was configured to reflect live updates, providing real-time insights into application data.

This document outlines the architecture, Terraform implementation, deployment flow, and key configurations that were used to meet the project requirements.

OBJECTIVE

- Build a production-ready infrastructure that includes:
- Auto Scaling EC2 instances with Nginx, Docker, and Node.js 20
- RDS databases (MySQL and PostgreSQL) in private subnets
- Load Balancer with HTTPS support
- Multi-stage Dockerized web applications (Frontend + Backend)
- BI Tool deployment (Metabase)
- Secure domain and SSL setup (ACM)
- SSH tunneling for database access
- Live dashboard visualization using Metabase

LINKS

React-App: https://github.com/MARN121/reactapp-devops

Terraform (IaC): https://github.com/MARN121/DevOps_Project

Project Structure (Modular)

Project/

- main.tf
- outputs.tf
- providers.tf
- terraform.tfvars
- variables.tf
- README.md
- modules/
  - network/
    - main.tf
    - outputs.tf
    - variables.tf
  - security_groups/
    - main.tf
    - outputs.tf
    - variables.tf
  - target_group/
    - main.tf
    - outputs.tf
    - variables.tf
  - ec2/
    - main.tf
    - outputs.tf
    - variables.tf
  - ec2-bi/
    - main.tf
    - outputs.tf
    - variables.tf
  - rds/
    - main.tf
    - outputs.tf
    - variables.tf
  - alb/
    - main.tf
    - outputs.tf
    - variables.tf
  - alb-bi/
    - main.tf
    - outputs.tf
    - variables.tf
  - route53/
    - main.tf
    - outputs.tf
    - variables.tf
- userdata/
  - userdata-app.sh
  - userdata-bi.sh
- docker/
  - Dockerfile
- snapshot-and-destroy.sh

PROJECT WALKTHROUGH

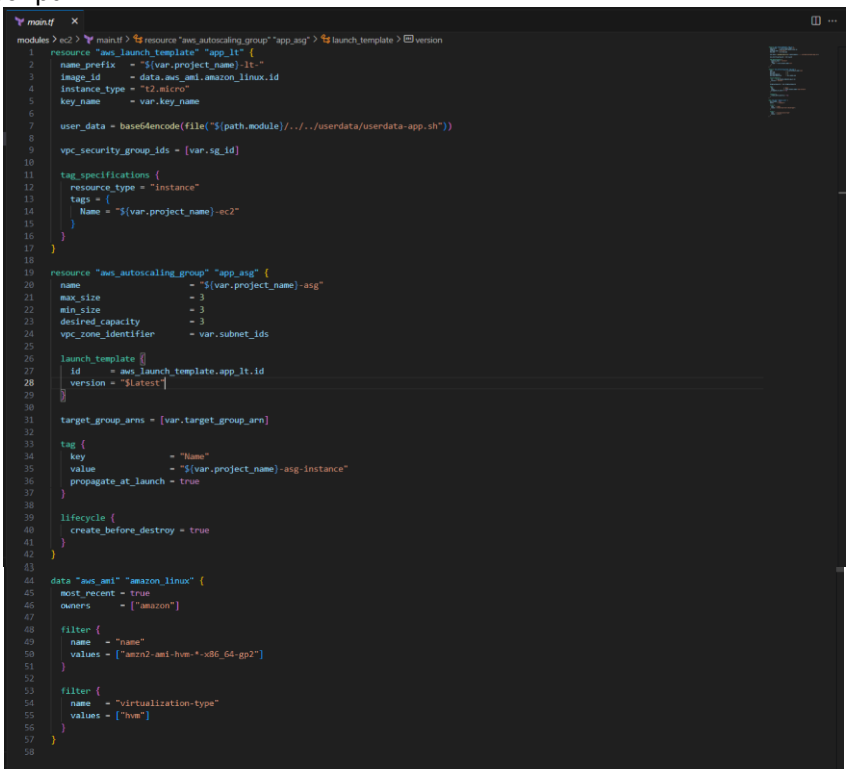
1. EC2 Auto Scaling Group

- 3 EC2 instances were launched using a Launch Template.
- User data scripts (*userdata-app.sh* and *userdata-bi.sh*) installed:
    - Nginx
    - Docker
    - Node.js 20

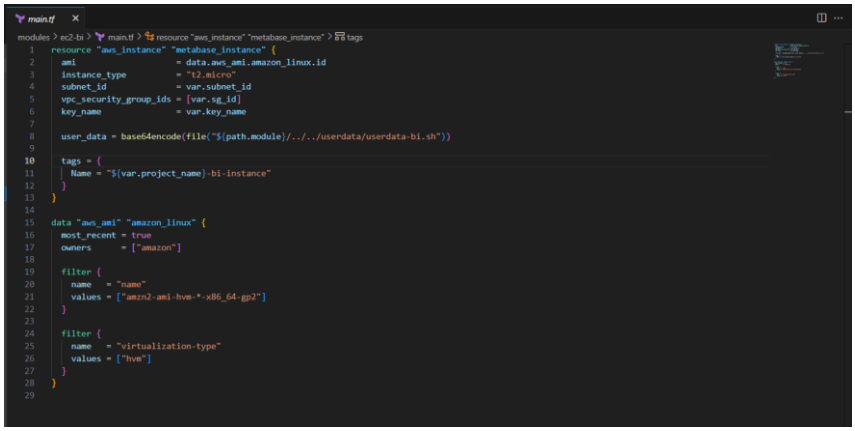
Three instances served frontend and backend application. And one instance hosted Metabase for BI visualization.

Screenshots:

Code/Script:



EC2 Main.tf (Application file)



EC2-BI Main.tf (BI file)

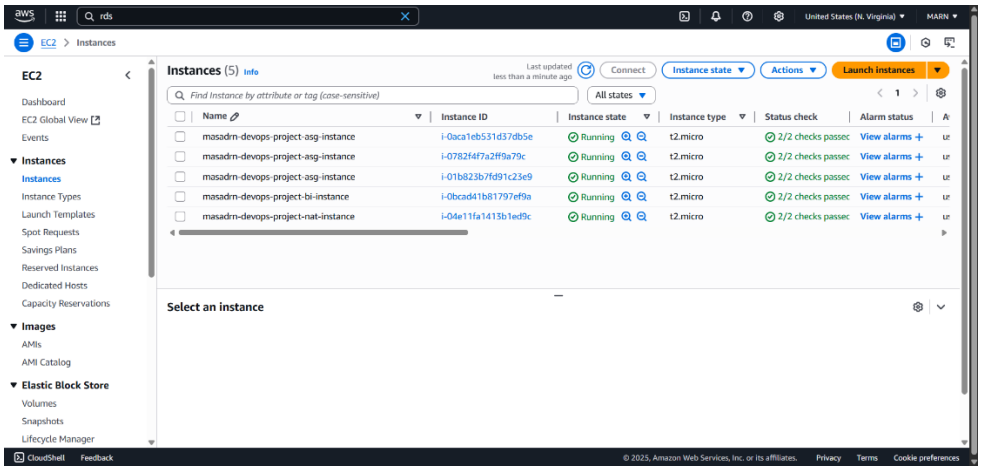
```
#!/bin/bash

# Update system packages
yum update -y

# Enable Amazon Linux extras for nginx and install required packages
amazon-linux-extras enable nginx1 -y
yum install -y nginx docker git

# Start and enable Docker service
systemctl enable docker --now
usermod -aG docker ec2-user

# Clone your monorepo (frontend + backend)
cd /home/ec2-user
git clone https://github.com/MARN121/reactapp-devops.git
chown -R ec2-user:ec2-user reactapp-devops
cd reactapp-devops

# --- FRONTEND SETUP ---
cd frontend
docker build -t frontend-app .
docker run -d -p 3000:80 --name frontend-app frontend-app
cd ..

# --- BACKEND SETUP ---
cd backend
docker build -t backend-app .
docker run -d -p 5000:5000 --name backend-app backend-app
cd ..

# --- NGINX CONFIGURATION ---
cat <<EOF > /etc/nginx/conf.d/frontend.conf
server {
    listen 80;
    server_name app.nendo.fun;

    location / {
        proxy_pass http://127.0.0.1:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade \$http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host \$host;
        proxy_cache_bypass \$http_upgrade;
    }
}
EOF

cat <<EOF > /etc/nginx/conf.d/backend.conf
server {
    listen 80;
    server_name appback.nendo.fun;

    location / {
        proxy_pass http://127.0.0.1:5000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade \$http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host \$host;
        proxy_cache_bypass \$http_upgrade;
    }
}
EOF

# Validate and restart NGINX
nginx -t && systemctl enable nginx --now && systemctl restart nginx
```

Userdata-app.sh (file)

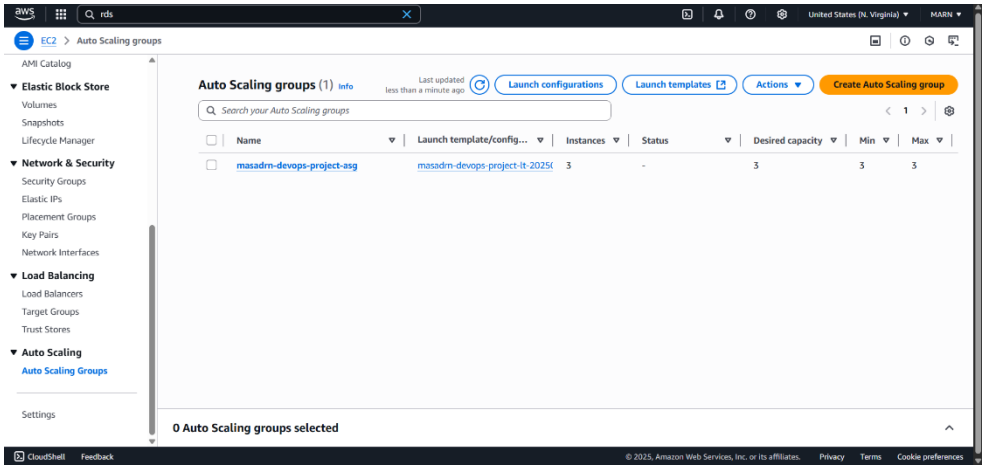
```
#!/bin/bash
yum update -y
amazon-linux-extras install docker -y
service docker start
usermod -aG docker ec2-user
docker run -d -p 3000:3000 --name metabase metabase/metabase
```

Userdata-bi.sh (file)

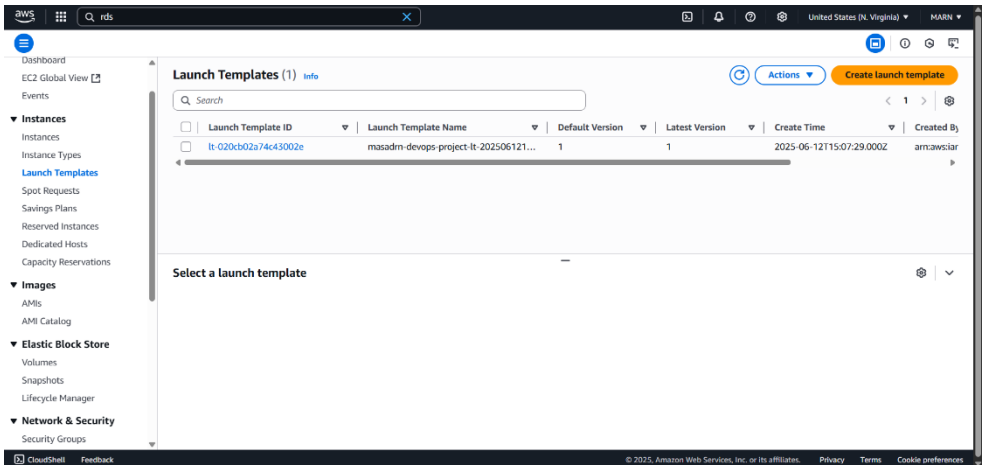
AWS:



Instances (AWS Console)



Auto Scaling Group (AWS Console)



Launch Template (AWS Console)

2. RDS Instances

- 1 MySQL and 1 PostgreSQL RDS instance created.
- Both RDS instances were launched in private subnets.
- Connected securely via SSH tunneling through EC2.
- Subnet groups and parameter groups configured via Terraform.

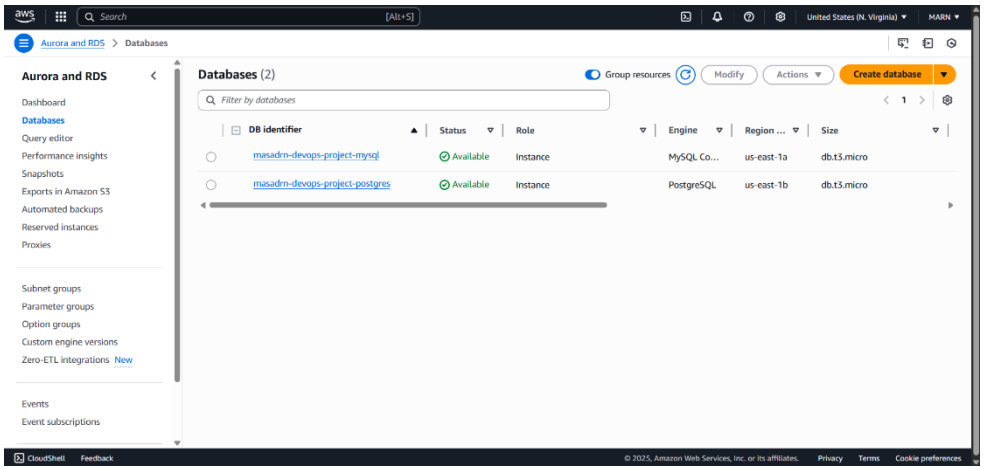
Screenshot:

Code/Script:

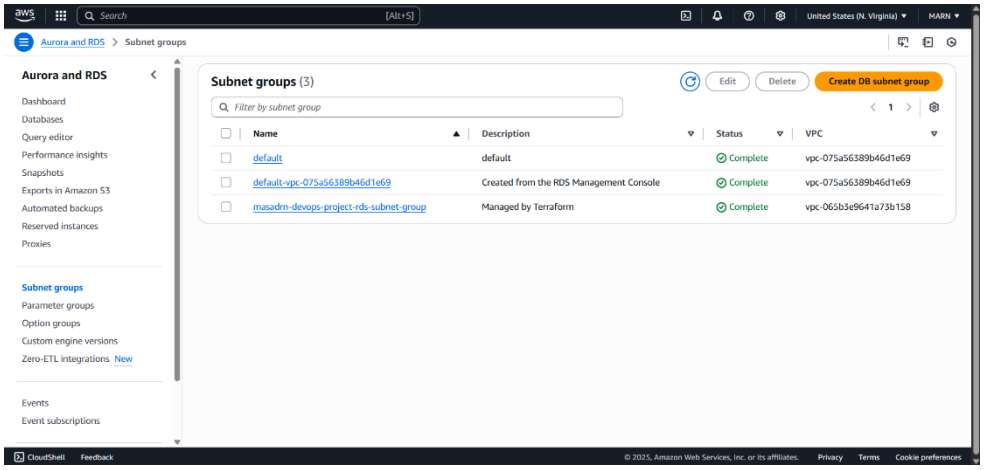
```hcl
# Subnet group for RDS
resource "aws_db_subnet_group" "rds_subnet_group" {
  name       = "${var.project_name}-rds-subnet-group"
  subnet_ids = var.subnet_ids
  tags = {
    Name = "${var.project_name}-rds-subnet-group"
  }
}

# MySQL Instance
resource "aws_db_instance" "mysql" {
  identifier             = "${var.project_name}-mysql"
  engine                 = "mysql"
  instance_class         = "db.t3.micro"
  allocated_storage      = 20
  username               = var.db_username
  password               = var.db_password
  skip_final_snapshot    = true
  publicly_accessible    = false
  db_subnet_group_name   = aws_db_subnet_group.rds_subnet_group.name
  vpc_security_group_ids = [var.rds_sg_id]
}

# PostgreSQL Instance
resource "aws_db_instance" "postgres" {
  identifier             = "${var.project_name}-postgres"
  engine                 = "postgres"
  instance_class         = "db.t3.micro"
  allocated_storage      = 20
  username               = var.db_username
  password               = var.db_password
  skip_final_snapshot    = true
  publicly_accessible    = false
  db_subnet_group_name   = aws_db_subnet_group.rds_subnet_group.name
  vpc_security_group_ids = [var.rds_sg_id]
}
```

RDS Main.tf (file)

AWS:



RDS Databases (AWS Console)



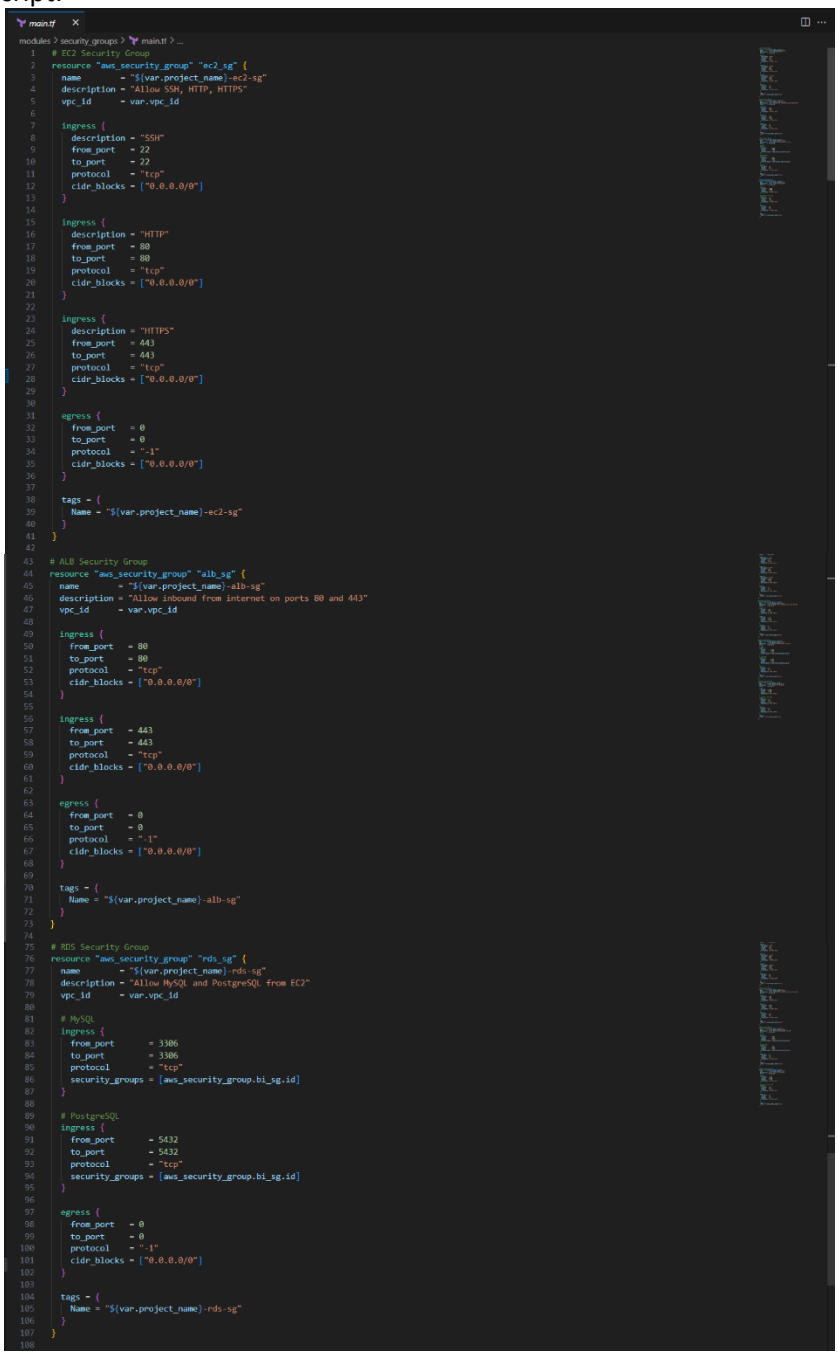
RDS Subnet-Groups (AWS Console)

3. Security Groups

- EC2 Security Group:
  - Ingress: Port 22 (SSH), 80 (HTTP), 443 (HTTPS)
- RDS Security Group:
  - Ingress: Ports 3306 (MySQL) and 5432 (PostgreSQL)
  - Access allowed only from EC2 security group
- ALB Security Group:
  - Ingress: Port 80 and 443 open to all
- Egress open to all (0.0.0.0/0) for all SGs.

Screenshot:

Code/Script:

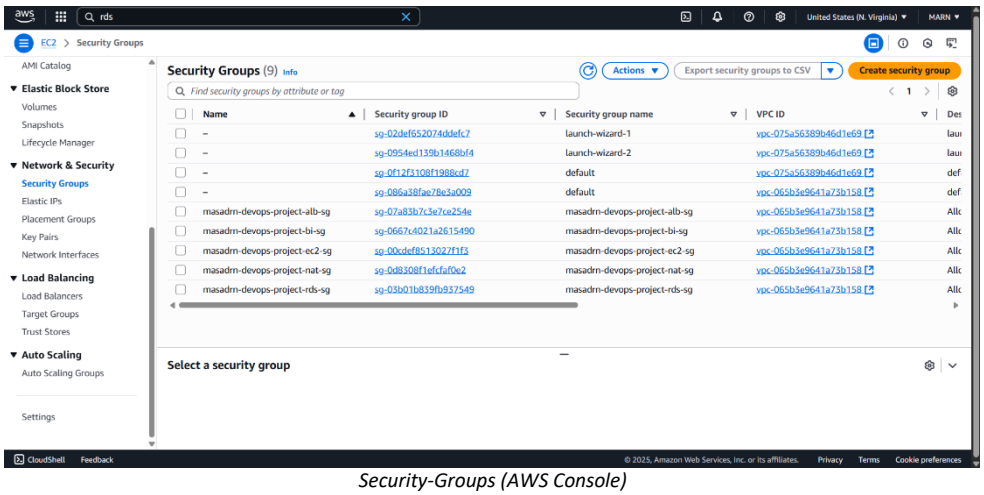


```
# EC2 Security Group
resource "aws_security_group" "ec2_sg" {
  name        = "${var.project_name}-ec2-sg"
  description = "Allow SSH, HTTP, HTTPS"
  vpc_id      = var.vpc_id

  ingress {
    description = "SSH"
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    description = "HTTP"
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    description = "HTTPS"
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "${var.project_name}-ec2-sg"
  }
}

# ALB Security Group
resource "aws_security_group" "alb_sg" {
  name        = "${var.project_name}-alb-sg"
  description = "Allow inbound from internet on ports 80 and 443"
  vpc_id      = var.vpc_id

  ingress {
    from_port   = 80
    to_port     = 80
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 443
    to_port     = 443
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "${var.project_name}-alb-sg"
  }
}

# RDS Security Group
resource "aws_security_group" "rds_sg" {
  name        = "${var.project_name}-rds-sg"
  description = "Allow MySQL and PostgreSQL from EC2"
  vpc_id      = var.vpc_id

  # MySQL
  ingress {
    from_port       = 3306
    to_port         = 3306
    protocol        = "tcp"
    security_groups = [aws_security_group.bi_sg.id]
  }

  # PostgreSQL
  ingress {
    from_port       = 5432
    to_port         = 5432
    protocol        = "tcp"
    security_groups = [aws_security_group.bi_sg.id]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "${var.project_name}-rds-sg"
  }
}
```

```hcl
# BI-EC2 Security Group
resource "aws_security_group" "bi_sg" {
  name        = "${var.project_name}-bi-sg"
  description = "Allow access for Metabase"
  vpc_id      = var.vpc_id

  ingress {
    from_port   = 3000
    to_port     = 3000
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  # Optional: Allow SSH
  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1"
    cidr_blocks = ["0.0.0.0/0"]
  }

  tags = {
    Name = "${var.project_name}-bi-sg"
  }
}
```

Security Group Main.tf (file)

AWS:



Security-Groups (AWS Console)

4. Load Balancer (ALB)

- Application Load Balancer deployed via Terraform.
- HTTPS enforced:
  - HTTP (port 80) redirected to HTTPS (port 443)
- Listener rules forward traffic to target groups based on subdomain.
- HTTPS configured with ACM certificate for secure access.

Screenshot:

Code/Script:

main.tf

modules > alb > main.tf > resource "aws_lb_listener" "https" > default_action > target_group_arn

```hcl
resource "aws_lb" "app_alb" {
  name               = "${var.project_name}-alb"
  load_balancer_type = "application"
  subnets            = var.subnet_ids
  security_groups    = [var.alb_sg_id]

  enable_deletion_protection = false
  idle_timeout               = 60

  tags = {
    Name = "${var.project_name}-alb"
  }
}

# HTTP → HTTPS Redirect
resource "aws_lb_listener" "http" {
  port              = 80
  protocol          = "HTTP"
  load_balancer_arn = aws_lb.app_alb.arn

  default_action {
    type = "redirect"
    redirect {
      port        = "443"
      protocol    = "HTTPS"
      status_code = "HTTP_301"
    }
  }
}

# HTTPS Listener
resource "aws_lb_listener" "https" {
  port              = 443
  protocol          = "HTTPS"
#   ssl_policy        = "ELBSecurityPolicy-TLS-1-2-Ext-2018-06"
  certificate_arn   = var.acm_certificate_arn
  load_balancer_arn = aws_lb.app_alb.arn

  default_action {
    type             = "forward"
    target_group_arn = var.target_group_arn
  }
}
```

ALB Main.tf (file)

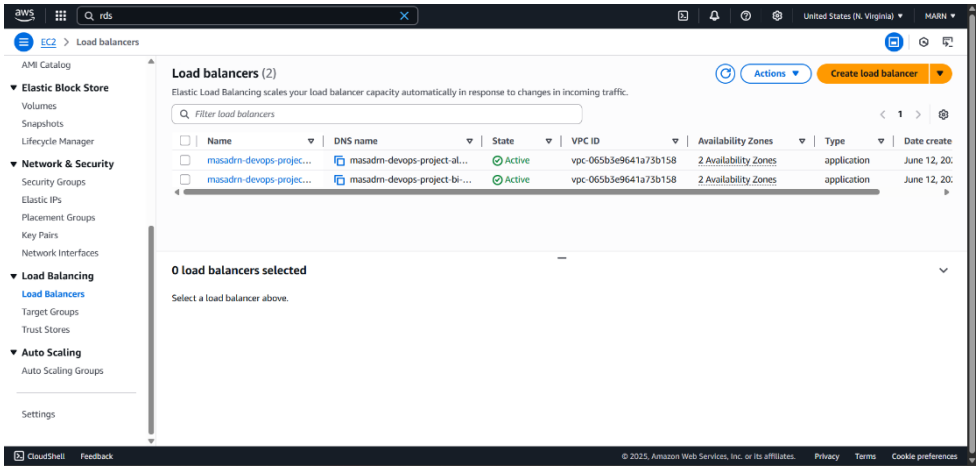
main.tf

modules > alb-bi > main.tf > resource "aws_lb" "bi_alb" > subnets

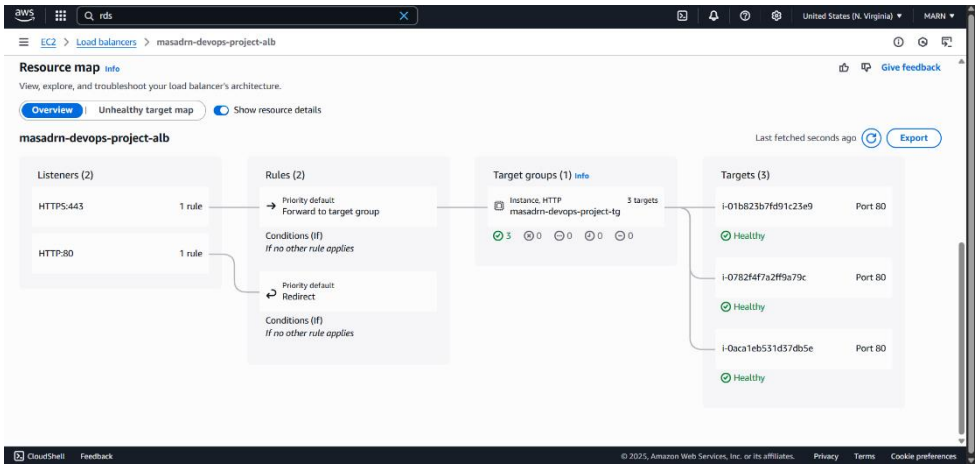
```hcl
resource "aws_lb" "bi_alb" {
  name               = "${var.project_name}-bi-alb"
  load_balancer_type = "application"
  subnets            = var.subnet_ids
  security_groups    = [var.alb_sg_id]
  enable_deletion_protection = false
  idle_timeout               = 60

  tags = {
    Name = "${var.project_name}-bi-alb"
  }
}

resource "aws_lb_target_group" "bi_tg" {
  name        = "${var.project_name}-bi-tg"
  port        = 3000
  protocol    = "HTTP"
  vpc_id      = var.vpc_id
  target_type = "instance"

  health_check {
    path                = "/"
    protocol            = "HTTP"
    port                = "3000"
    interval            = 30
    timeout             = 5
    healthy_threshold   = 2
    unhealthy_threshold = 2
  }

  tags = {
    Name = "${var.project_name}-bi-tg"
  }
}

resource "aws_lb_target_group_attachment" "bi_attach" {
  target_group_arn = aws_lb_target_group.bi_tg.arn
  target_id        = var.instance_id
  port             = 3000
}

resource "aws_lb_listener" "http" {
  load_balancer_arn = aws_lb.bi_alb.arn
  port              = 80
  protocol          = "HTTP"

  default_action {
    type = "redirect"
    redirect {
      port        = "443"
      protocol    = "HTTPS"
      status_code = "HTTP_301"
    }
  }
}

resource "aws_lb_listener" "https" {
  load_balancer_arn = aws_lb.bi_alb.arn
  port              = 443
  protocol          = "HTTPS"
  ssl_policy        = "ELBSecurityPolicy-2016-08"
  certificate_arn   = var.acm_certificate_arn

  default_action {
    type             = "forward"
    target_group_arn = aws_lb_target_group.bi_tg.arn
  }
}
```

ALB-BI Main.tf (file)

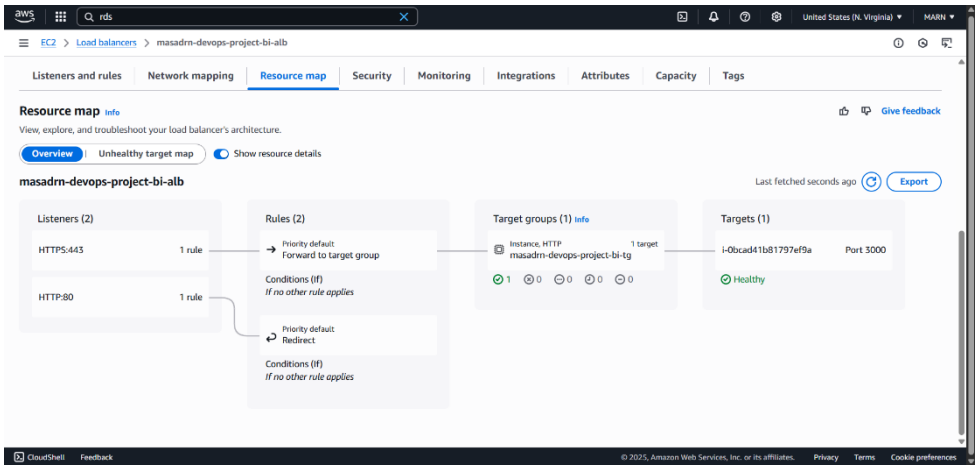
AWS:



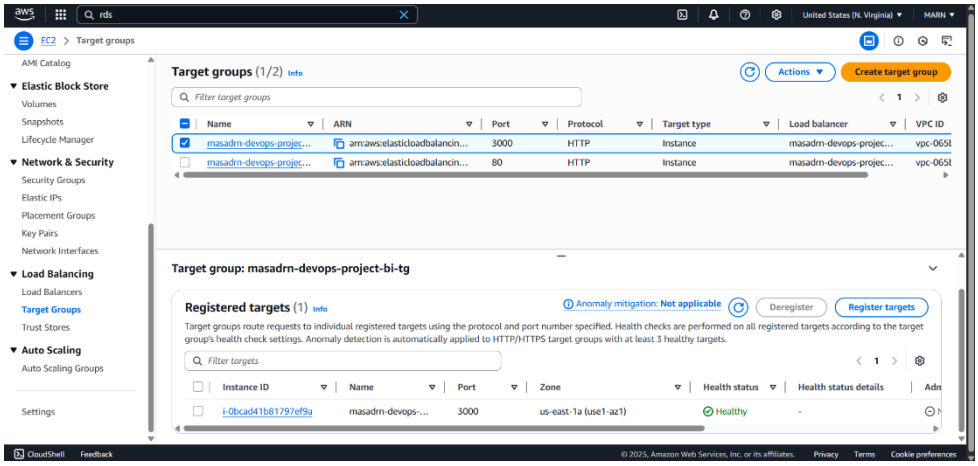
Load Balancers (AWS Console)



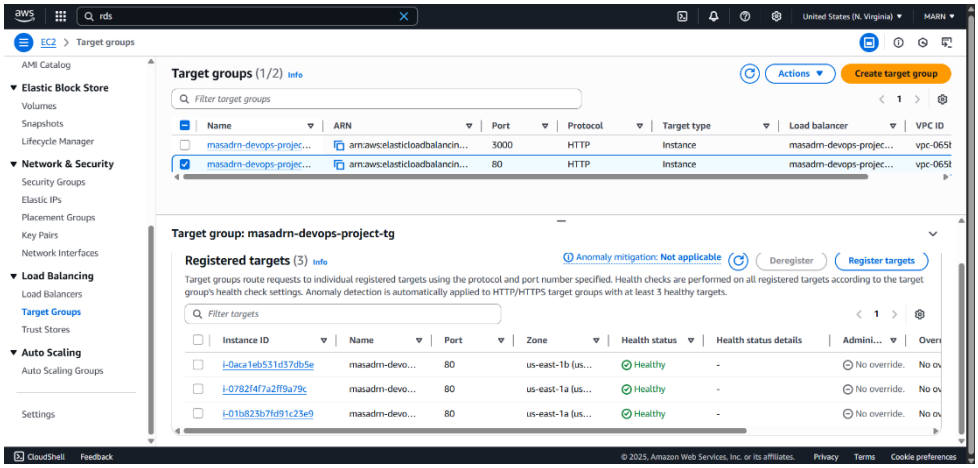
Application Load Balancer – Resource Map (AWS Console)



BI Load Balancer – Resource Map (AWS Console)



BI Target Group, with Instance Health (AWS Console)



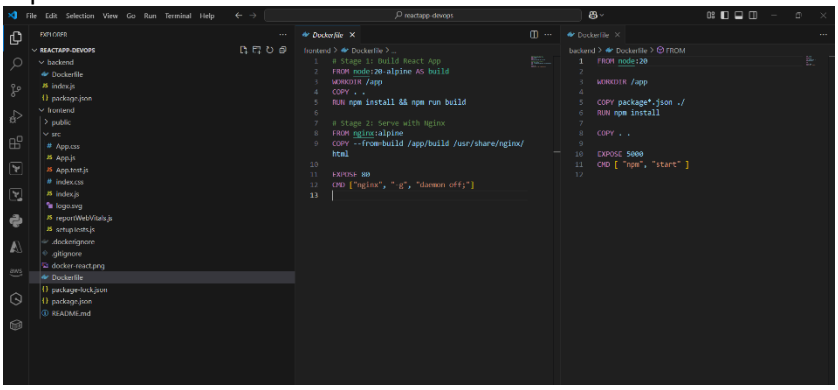
Application Target Group, with Instance Health (AWS Console)

5. Dockerized App Deployment

- Used multi-stage Dockerfiles for React frontend and Nomral for Node backend (Comparision).
- Deployed both apps on same EC2 using Docker:
  - Frontend: app.nendo.fun (port 3000 → NGINX → 80)
  - Backend: appback.nendo.fun (port 5000 → NGINX → 80)
- NGINX set up as reverse proxy for each app.

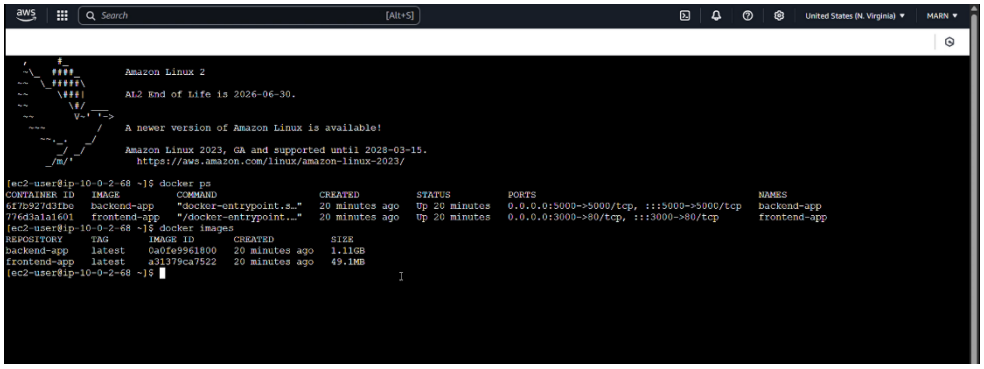
Screenshot:

Code/Script:



React Application Frontend (Multi-Stage) and Backend (Normal) Dockerfile (file)

AWS:



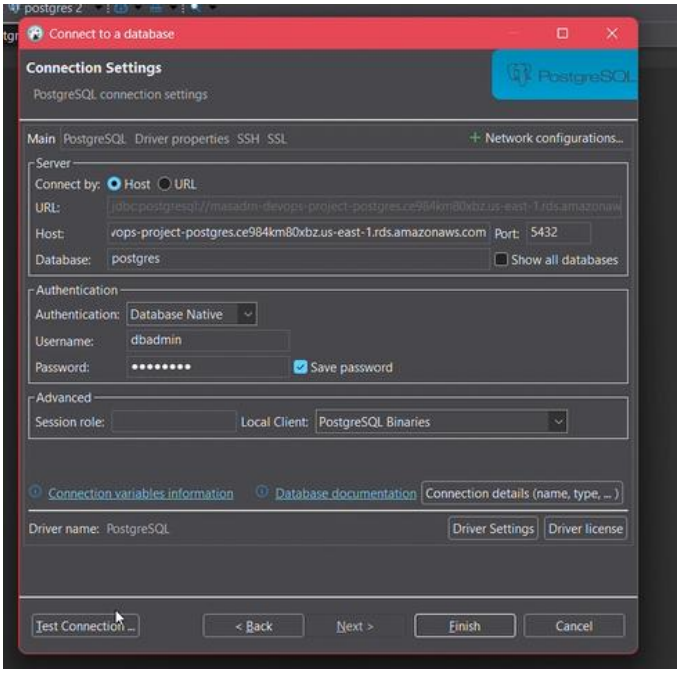
EC2 Instance SSH terminal via Instance Connect (AWS Console)
Docker Containers Running and Image Size Comparison

6. Database Access & Dummy Data

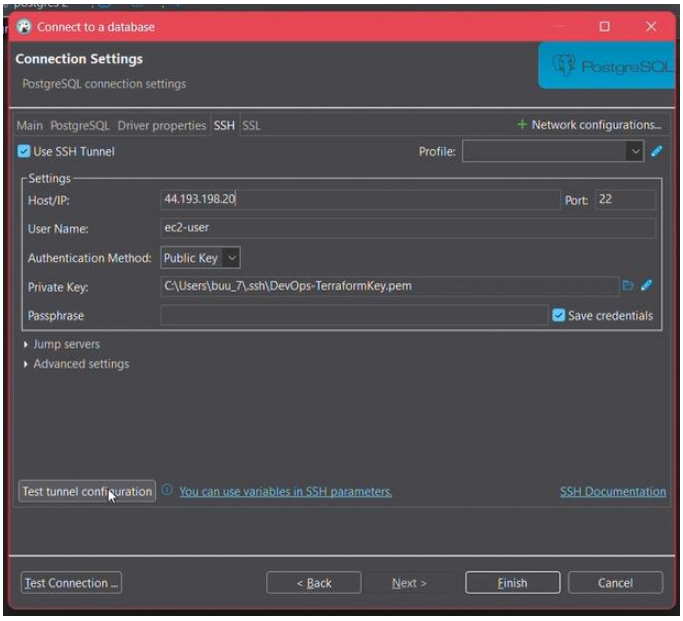
- Used DBeaver to connect to RDS PostgreSQL via SSH tunnel through EC2.
- Dummy data added to a sales table:
  - Console sales for PlayStation, Xbox, Steam Deck, etc.
- Verified live updates in Metabase dashboard after inserting records.

Screenshot:

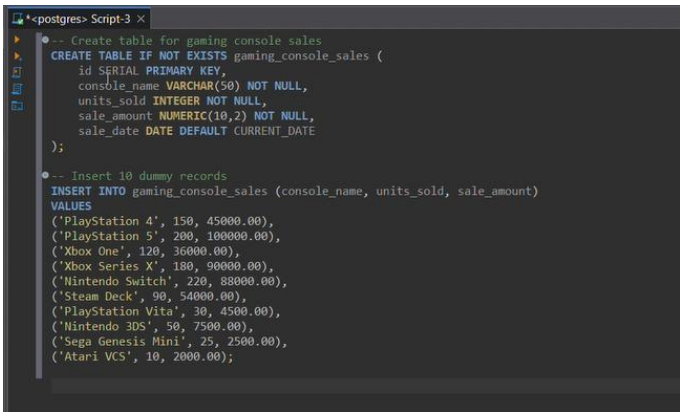
DBeaver:



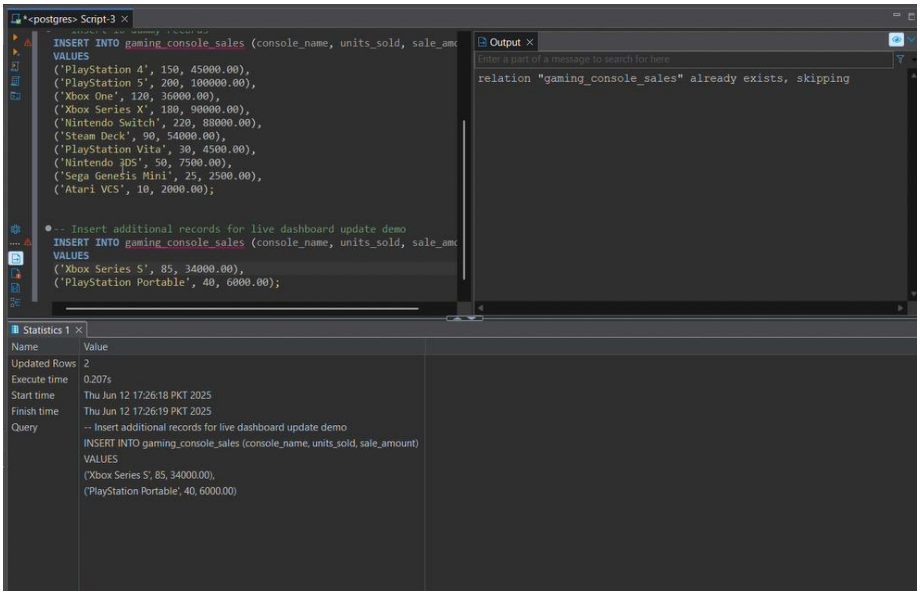
Connecting to our deployed Database on DBeaver by setting up Main and SSH settings.
Host will be our DB Endpoint provided by AWS RDS
Username and Password is set by us in TF-variables.



SSH tunneling because RDS are not deployed publicly.
Host is EC2 Instance public IP on which RDS are deployed
Username will be ec2-user (default user of instance)
For authentication we will use key .pem file of key-pair we are using in Terraform

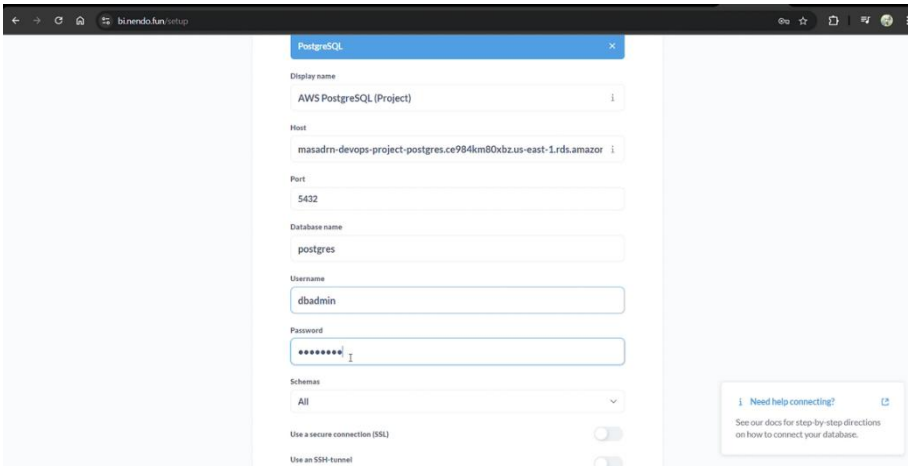


Dummy Data Script to create table and add dummy records.

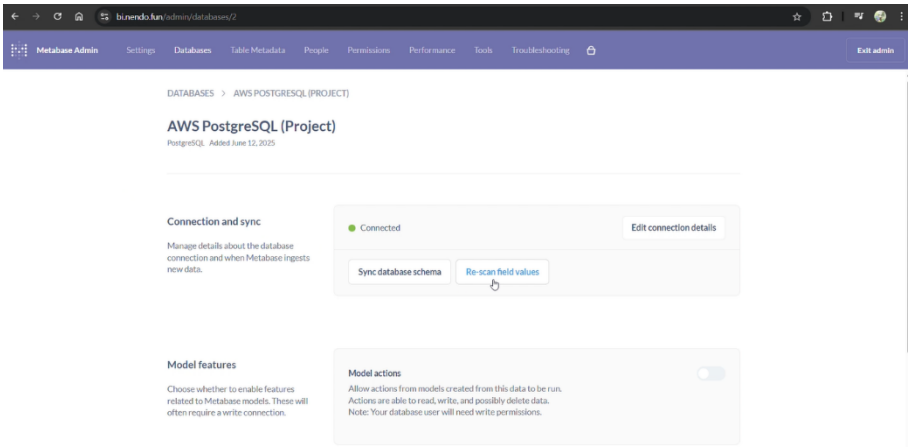


Dummy Data Script Insert with Showing Result Successful and Outputting Table already exists on running same script again.

Metabase:



Using DB endpoint we will setup Database on Metabase
Username and Password is set by us in TF-variables.



Sync and Scan Fields after adding updating DB First Time

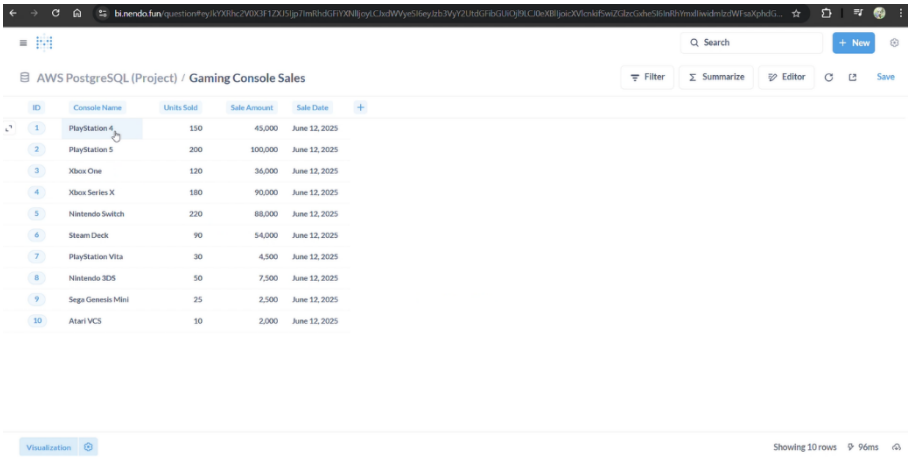


Table Before Update

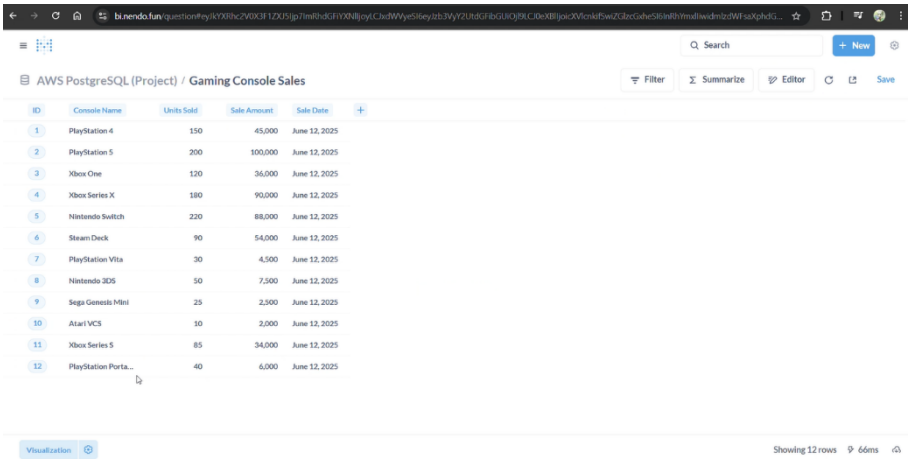
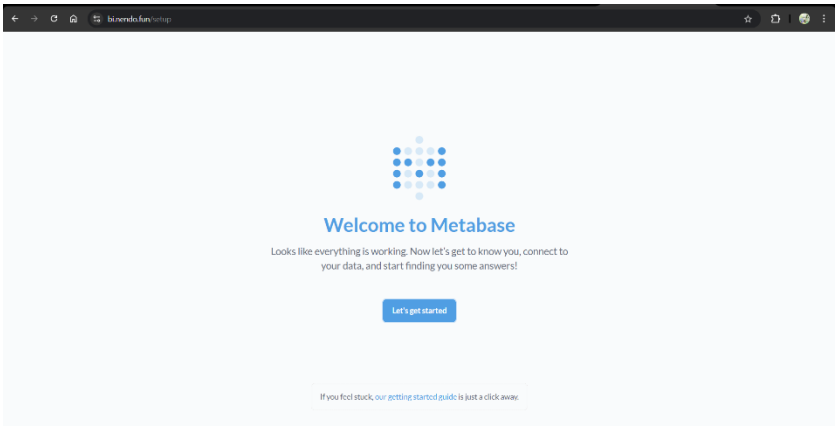


Table After Update

7. BI Tool Deployment (Metabase)

- Deployed Metabase using Docker on a dedicated EC2 instance.
- Available via bi.nendo.fun, secured with HTTPS.
- Connected to PostgreSQL RDS to visualize sales data.
- Dashboard created to reflect real-time data.

Screenshot:



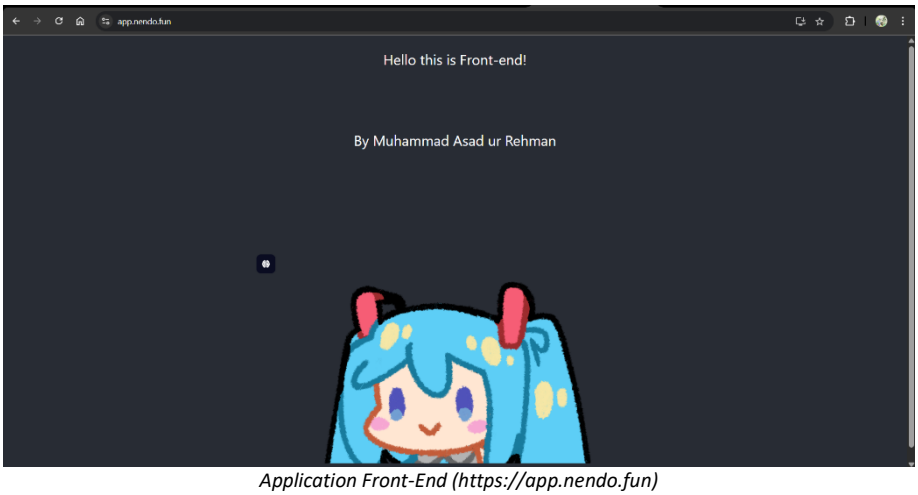
Metabase deployed on HTTPS://bi.nendo.fun (secure)

Metabase Deployment in part 1
Updates and data display in part 6

8. Domain & SSL Configuration

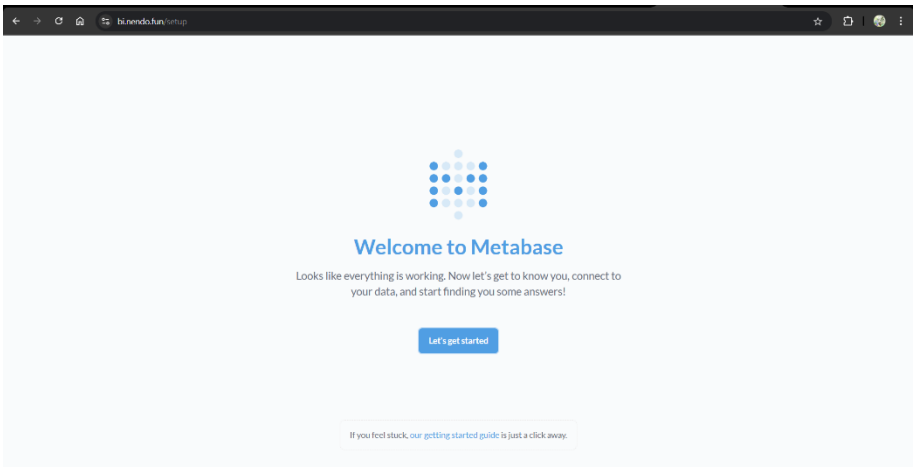
- Hosted zone: nendo.fun managed in Route 53.
- Subdomains routed using ALB DNS:
  - app.nendo.fun → Frontend
  - appback.nendo.fun → Backend
  - bi.nendo.fun → BI (Metabase)
- ACM certificate issued and validated for:
  - *.nendo.fun
  - nendo.fun

Screenshot:



Application Front-End (https://app.nendo.fun)

Application Back-End (https://appback.nendo.fun)



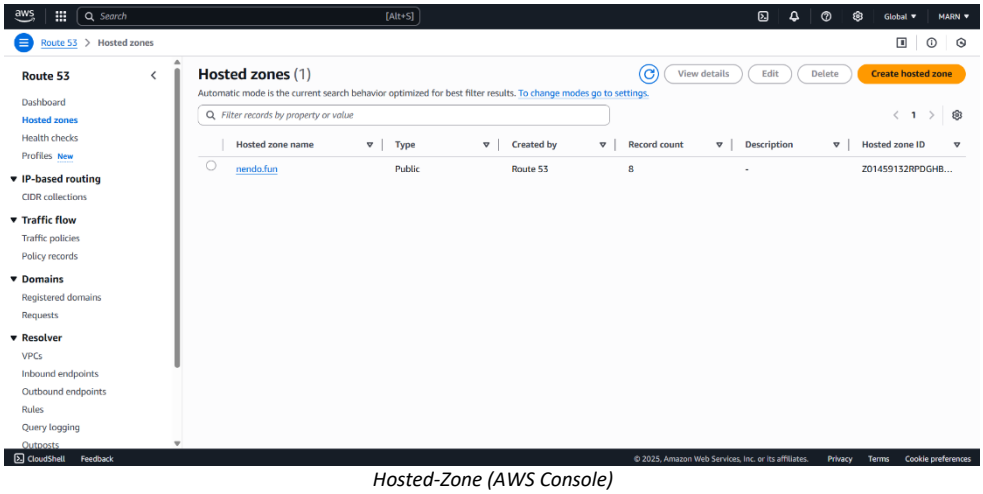
Metabase (https://bi.nendo.fun)

Code/Script:

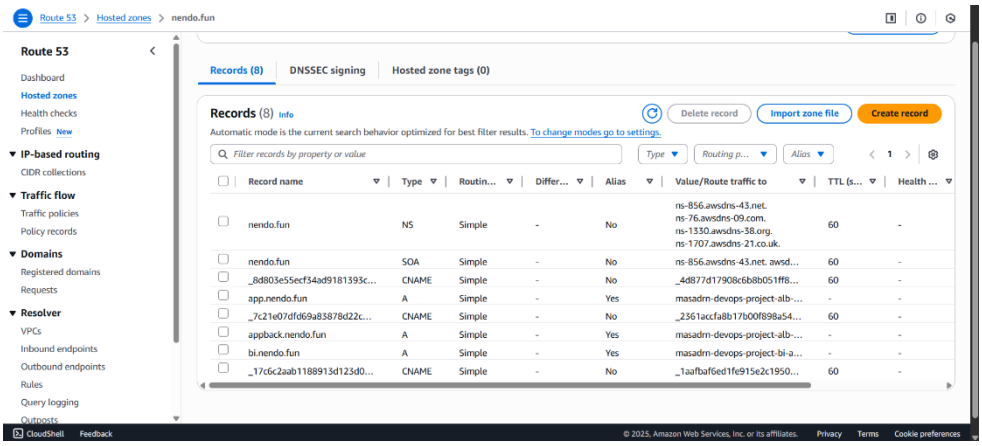
```hcl
# resource "aws_route53_zone" "main" {
#   name = var.domain_name  # "nendo.fun"
# } # New Hosted Zone using TF

resource "aws_route53_record" "app" {
  zone_id = var.hosted_zone_id
  name = "app.${var.domain_name}"  # app.nendo.fun
  type = "A"

  alias {
    name = var.alb_dns_name
    zone_id = var.alb_zone_id  # us-east-1
    evaluate_target_health = true
  }
}

resource "aws_route53_record" "appback" {
  zone_id = var.hosted_zone_id
  name    = "appback.${var.domain_name}"
  type    = "A"

  alias {
    name                   = var.alb_dns_name
    zone_id                = var.alb_zone_id
    evaluate_target_health = true
  }
}

# A Record for BI Tool – EC2 instance (optional for Metabase)
resource "aws_route53_record" "bi" {
  zone_id = var.hosted_zone_id
  name    = "bi.${var.domain_name}" # bi.nendo.fun
  type    = "A"

  alias {
    name = var.bi_alb_dns_name
    zone_id = var.alb_zone_id
    evaluate_target_health = true
  }
}
```

RDS Main.tf (file)

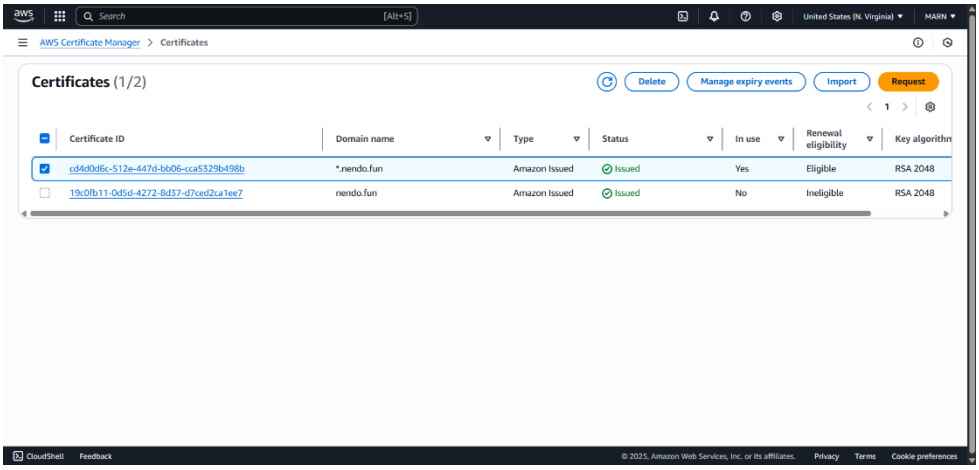
AWS:



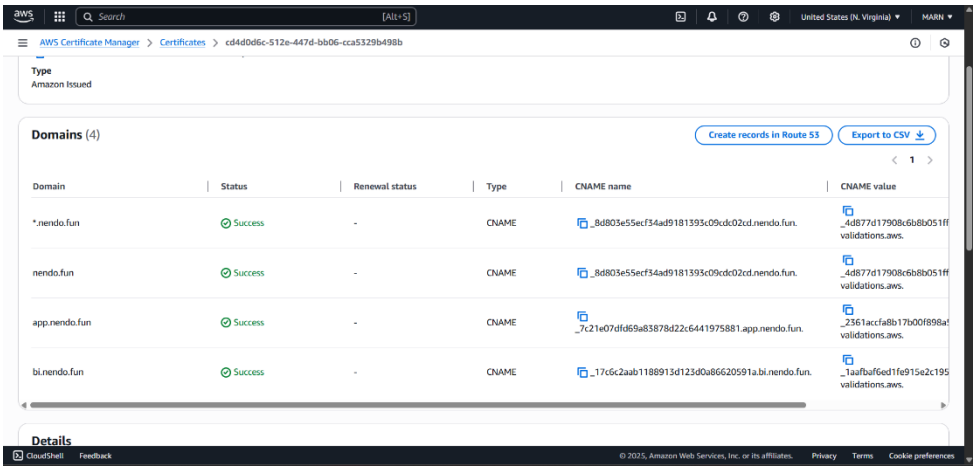
Hosted-Zone (AWS Console)



Hosted-Zone Records (AWS Console)



ACM (AWS Console)



ACM Domains (AWS Console)

9. Snapshot Strategy (Extra)

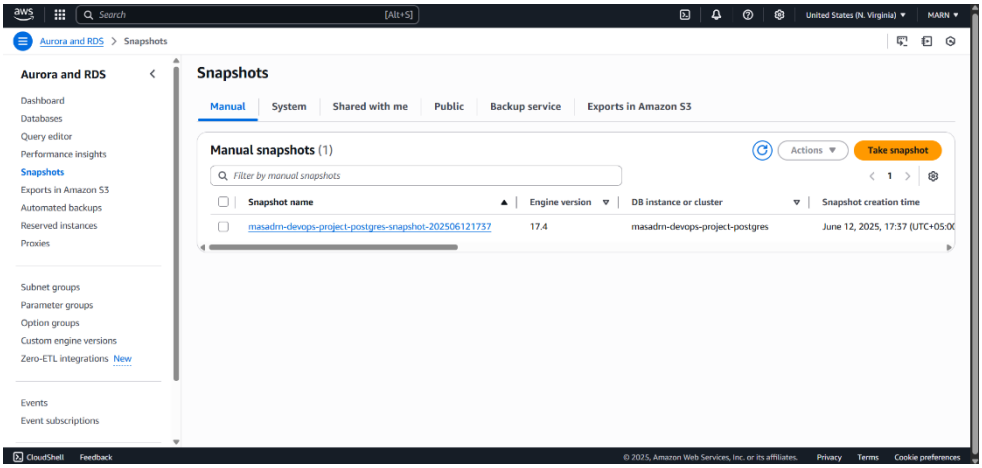
- Shell script snapshot-and-destroy.sh created.
- Before destroying Terraform infra:
  - Takes RDS snapshots to preserve data and user config.
  - Then runs terraform destroy for clean teardown.
- Can restore from snapshot on re-deploy.

Screenshot:

Code/Script:

Snapshot script (file)

AWS:



RDS Snapshot (AWS Console)