Infraestructura de eCommerce en AWS: Configuración de Amazon S3

Valeria Pillimue, Sergio Colmenares, Juan Canizales

28 de noviembre de 2024

Índice

1 Introducción

Este documento describe la implementación de un sistema de comercio electrónico (*eCommerce*) en Amazon Web Services (AWS). El objetivo principal es diseñar una infraestructura robusta, escalable y segura, dirigida a pequeñas y medianas empresas (PyMEs) que enfrentan limitaciones de recursos tecnológicos.

Entre los componentes fundamentales del sistema se encuentra **Amazon S3**, que servirá como el servicio de almacenamiento de archivos estáticos, tales como imágenes de productos y recursos relacionados.

2 Configuración de Amazon S3

2.1 Especificaciones del Bucket

El bucket configurado para el almacenamiento de imágenes estáticas del sistema eCommerce incluye los siguientes parámetros:

- **Nombre del bucket**: `ecommerce-store-images`, asegurando unicidad global dentro de AWS.
- **Región**: `US East (N. Virginia)`, optimizando costos y tiempos de respuesta.
- **Tipo de bucket**: *General Purpose*, adecuado para accesos regulares y redundancia entre zonas de disponibilidad.
- **Propiedad de objetos**: Se seleccionó *Bucket owner enforced*, restringiendo el control total a la cuenta propietaria.
- **Versionado**: Activado para permitir la recuperación de versiones anteriores de los archivos en caso de modificaciones o eliminaciones accidentales.
- **Cifrado**: Implementado con *Server-side encryption with Amazon S3 managed keys (SSE-S3)*, asegurando que los datos estén protegidos durante el almacenamiento.

2.2 Integración con el Sistema

El bucket será utilizado por el backend del eCommerce para:

- Subir y gestionar imágenes de productos.
- Permitir acceso dinámico a través de políticas de permisos para ciertas imágenes públicas.
- Integrarse mediante el SDK de AWS con los servicios relacionados.

2.3 Política de Acceso Pública

La política de acceso configurada para el bucket `ecommerce-store-images` permite el acceso de solo lectura a todos los usuarios públicos, lo cual es adecuado para recursos como imágenes de productos en el sistema *eCommerce*.

Política JSON:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::ecommerce-store-images/*"
        }
    ]
}
```

Descripción de la Política:

- **Version**: Especifica el formato de la política, definido como `"2012-10-17"`.
- **Sid**: Identificador único de la política, en este caso `"PublicReadGetObject"`.
- **Effect**: Establece que la acción permitida es `."llow"`.
- **Principal**: Define que todos los usuarios públicos (`"*"`) tienen acceso.
- **Action**: La acción permitida es `"s3:GetObject"`, que habilita la lectura de los objetos almacenados.
- **Resource**: Especifica que la política aplica a todos los objetos dentro del bucket `ecommerce-store-images`.

2.4 Subida de Imágenes a S3

Para la subida de imágenes al bucket de Amazon S3, se desarrolló un script en Python que permite cargar imágenes de manera más directa. Este script utiliza la biblioteca `boto3` para interactuar con los servicios de AWS.

Requisitos:

- Tener configurado el cliente AWS CLI con credenciales válidas.
- El bucket `ecommerce-store-images` debe existir previamente.

- Python 3 instalado junto con la biblioteca `boto3`.

Código del Script:

```python
import boto3
import os
import sys

s3 = boto3.client('s3')
NOMBRE_BUCKET = 'ecommerce-store-images'
CARPETA_S3 = 'uploads/'

for ruta_archivo in sys.argv[1:]:
    if os.path.isfile(ruta_archivo):
        nombre_archivo = os.path.basename(ruta_archivo)
        clave_s3 = os.path.join(CARPETA_S3, nombre_archivo)
        s3.upload_file(ruta_archivo, NOMBRE_BUCKET, clave_s3)
        print(f"Archivo subido: {ruta_archivo} a s3://{NOMBRE_BUCKET}/{clave_s3}")
    else:
        print(f"No es un archivo v lido: {ruta_archivo}")
```

Listing 1: Script de subida de imágenes a S3

```
(myenv) C:\Users\Cybor\Desktop\aws>python src\subir_a_s3.py C:\Users\Cybor\Downloads\1000x800.png
Archivo subido: C:\Users\Cybor\Downloads\1000x800.png a s3://ecommerce-store-images/uploads/1000x800.png
```

3 Configuración de Amazon RDS

Para gestionar la información del eCommerce, se configuró una base de datos utilizando el servicio Amazon RDS (*Relational Database Service*). Este servicio permite gestionar una base de datos escalable y segura con un mínimo de administración, aprovechando el modelo *managed* de AWS.

3.1 Detalles de Configuración

La base de datos fue configurada utilizando la opción *Easy Create*, con los siguientes parámetros:

- **Motor de Base de Datos**: MySQL.
- **Nivel de Uso**: Free Tier, ideal para desarrollo y pruebas.
- **Identificador de la Base de Datos**: `ecommerce-db`.
- **Gestión de Credenciales**: Administradas en AWS Secrets Manager.

3.2 Parámetros de Conexión

Una vez creada la base de datos, se obtuvieron los siguientes parámetros clave desde la consola de AWS:

- **Endpoint**: Dirección única de la base de datos proporcionada por RDS.
- **Puerto**: 3306 (puerto predeterminado para MySQL).
- **Usuario Maestro**: `admin`.
- **Credenciales**: Gestionadas automáticamente por AWS Secrets Manager.

3.3 Estructura de la Base de Datos

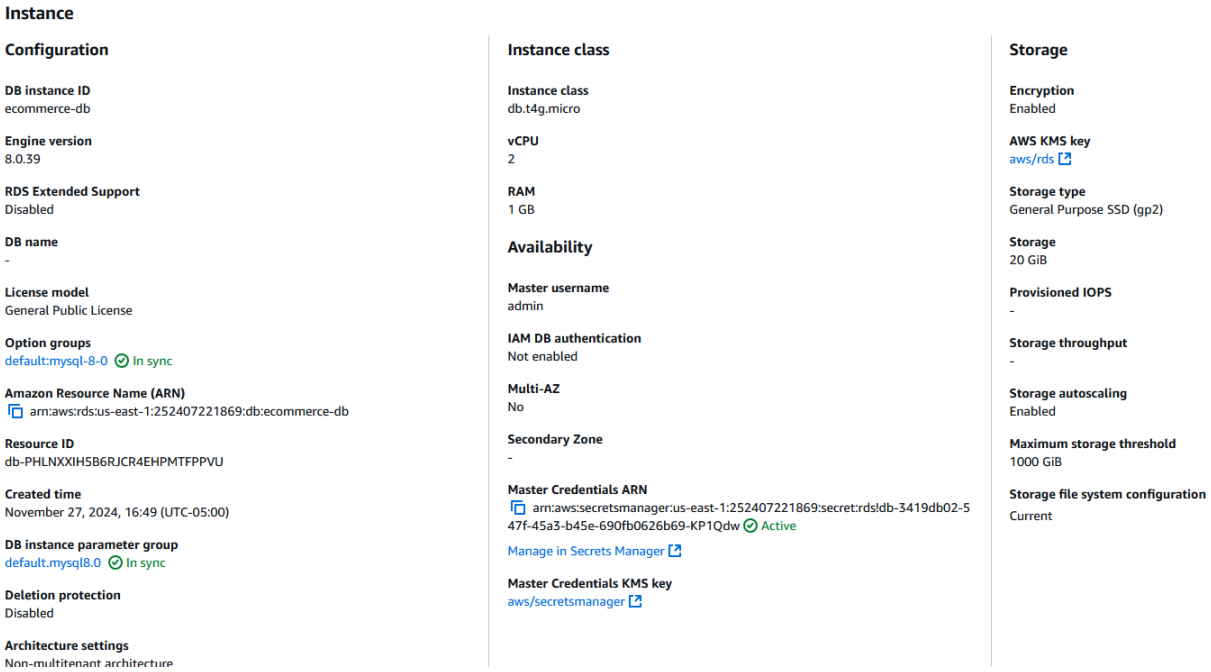
Para este sistema de eCommerce, se diseñaron las siguientes tablas iniciales:

3.3.1 Tabla `products`

Encargada de almacenar los detalles de los productos disponibles en la tienda.

```
CREATE TABLE products (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    description TEXT,
    image_url VARCHAR(255) NOT NULL
);
```

Listing 2: Definición de la tabla `products`



3.4 Gestión de la Base de Datos para el eCommerce

3.4.1 Configuración en AWS

Se utilizaron los siguientes servicios de AWS:

- **Amazon Secrets Manager**: Proporciona credenciales seguras para conectarse a la base de datos RDS
- **RDS Security Group**: Se configuró para aceptar conexiones en el puerto 3306 exclusivamente desde la dirección IP del cliente local con inbound rules para MySQL.

3.4.2 Conexión a Amazon RDS

Para conectarse a la base de datos `ecommerce_db` alojada en Amazon RDS, se implementó la siguiente función:

```python
def get_db_connection(secret, use_db=True):
    return pymysql.connect(
        host="ecommerce-db.cdj5i4puctle.us-east-1.rds.amazonaws.com",
        user=secret["username"],
        password=secret["password"],
        database="ecommerce_db" if use_db else None,
        port=3306
    )
```

Listing 3: Conexión a Amazon RDS

3.4.3 Sentencias SQL Implementadas

A continuación, se muestran las principales operaciones SQL para gestionar la base de datos:

Crear Base de Datos La base de datos `ecommerce_db` se crea solo si no existe, utilizando la siguiente función:

```
def create_database_if_not_exists(secret):
    connection = get_db_connection(secret, use_db=False)
    try:
        with connection.cursor() as cursor:
            cursor.execute("CREATE␣DATABASE␣IF␣NOT␣EXISTS␣ecommerce_db;
                ")
    finally:
        connection.close()
```

Listing 4: Crear Base de Datos

Crear Tabla de Productos Se define una tabla llamada `products` que almacena los detalles de los productos:

```
def create_table_if_not_exists(secret):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("""
                CREATE TABLE IF NOT EXISTS products (
                    id SERIAL PRIMARY KEY,
                    name VARCHAR(100) NOT NULL,
                    price DECIMAL(10, 2) NOT NULL,
                    description TEXT,
                    image_url VARCHAR(255) NOT NULL
                );
            """)
            connection.commit()
    finally:
        connection.close()
```

Listing 5: Crear Tabla de Productos

Insertar Productos Agrega nuevos productos con nombre, precio, descripción y la URL de la imagen:

```
def insert_product_to_db(secret, name, price, description, image_url):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("""
                INSERT INTO products (name, price, description,
                    image_url)
                VALUES (%s, %s, %s, %s);
```

```
            """, (name, price, description, image_url))
            connection.commit()
    finally:
        connection.close()
```

Listing 6: Insertar Productos

Consultar Productos Devuelve todos los productos almacenados en la tabla:

```
def fetch_all_products(secret):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("SELECT * FROM products;")
            rows = cursor.fetchall()
            return rows
    finally:
        connection.close()
```

Listing 7: Consultar Productos

Eliminar Productos Elimina un producto especificado por su `ID`:

```
def delete_product(secret, product_id):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("DELETE FROM products WHERE id = %s;", (
                product_id,))
            connection.commit()
    finally:
        connection.close()
```

Listing 8: Eliminar Productos

3.4.4 Resumen de la Gestión

El sistema implementado realiza las siguientes operaciones principales:

- Conexión segura con Amazon RDS mediante `Secrets Manager`.
- Creación y mantenimiento de la base de datos y tabla de productos.
- Inserción, consulta y eliminación de productos a través de sentencias SQL.

3.4.5 Insertar Producto con Imagen Asociada

El sistema permite asociar un enlace a la imagen almacenada en S3 con un producto en la base de datos. La inserción incluye los campos: nombre, precio, descripción y URL de la imagen. El siguiente código implementa esta funcionalidad:

```python
def insert_product_to_db(secret, name, price, description, image_url):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("""
                INSERT INTO products (name, price, description,
                    image_url)
                VALUES (%s, %s, %s, %s);
            """, (name, price, description, image_url))
            connection.commit()
    finally:
        connection.close()
```

Listing 9: Insertar Producto con Enlace a Imagen

Explicación del Código

- **Conexión a la Base de Datos**: Se utiliza la función `get_db_connection(secret)` para establecer una conexión segura con la base de datos `ecommerce_db`.
- **Sentencia SQL**: La instrucción `INSERT INTO` agrega un nuevo producto con sus detalles:
  - `name`: Nombre del producto.
  - `price`: Precio del producto (tipo decimal).
  - `description`: Descripción del producto.
  - `image_url`: Enlace a la imagen, previamente subida al bucket de S3.
- **Commit**: Confirma la transacción para asegurar que el registro se guarde en la base de datos.

Integración con S3 El sistema utiliza la función `upload_to_s3` para subir la imagen del producto al bucket S3 y generar el enlace URL correspondiente. Este enlace es luego asociado al producto en la base de datos.

```python
def upload_to_s3(file_path):
    s3 = boto3.client('s3')
    file_name = os.path.basename(file_path)
    s3_key = os.path.join(CARPETA_S3, file_name)
    s3.upload_file(file_path, BUCKET_NAME, s3_key)
    return f"https://{BUCKET_NAME}.s3.{REGION_NAME}.amazonaws.com/{
        s3_key}"
```

Listing 10: Subir Imagen y Generar URL

Resumen del Flujo

1. El usuario proporciona los datos del producto y la ruta de la imagen.
2. La imagen se sube al bucket S3, generando un URL público.
3. El producto, junto con el URL de la imagen, se inserta en la base de datos.

```
Opciones:
1. Agregar un producto
2. Consultar todos los productos
3. Eliminar un producto
4. Salir
Selecciona una opción: 1
Ruta de la imagen del producto: C:\Users\Cybor\Downloads\nerfgun.jpg
Nombre del producto: Pistola Nerf
Precio del producto: 200000
Descripción del producto: Nerf Dual-Strike Gun
Producto 'Pistola Nerf' agregado exitosamente.

Opciones:
1. Agregar un producto
2. Consultar todos los productos
3. Eliminar un producto
4. Salir
Selecciona una opción: 2
ID: 2, Nombre: Pistola Nerf, Precio: 200000.00, Descripción: Nerf Dual-Strike Gun, Imagen: https://ecommerce-store-images.s3.us-east-1.amazonaws.com/products/nerfgun.jpg
```

4 Desarrollo de la Aplicación Web con Flask

Para transformar la aplicación de línea de comandos en una aplicación web interactiva y visualmente atractiva, se utilizó el framework **Flask**. Flask es un framework ligero de Python que facilita el desarrollo de aplicaciones web de manera rápida y eficiente. A continuación, se detallan los aspectos clave de la implementación de la aplicación web.

4.1 Estructura del Proyecto

La estructura del proyecto se organizó de manera modular para facilitar el mantenimiento y la escalabilidad. La organización básica del proyecto es la siguiente:

- **app.py**: Archivo principal que inicializa la aplicación Flask y define las rutas.
- **requirements.txt**: Archivo que lista todas las dependencias necesarias para el proyecto.
- **templates/**: Directorio que contiene las plantillas HTML utilizadas para renderizar las páginas web.
- **static/**: Directorio que almacena los archivos estáticos como CSS, imágenes y scripts JavaScript.
- **config/**: (Opcional) Directorio para archivos de configuración adicionales.

Esta organización permite separar claramente la lógica de la aplicación, las vistas y los recursos estáticos, facilitando futuras expansiones y modificaciones.

4.2 Configuración del Entorno

Para garantizar un entorno de desarrollo consistente y reproducible, se configuró un entorno virtual utilizando **venv**. Esto aísla las dependencias del proyecto de otras aplicaciones en el sistema.

- **Creación y activación del entorno virtual**: Se utilizó el módulo `venv` de Python para crear un entorno virtual y activarlo, asegurando que todas las dependencias se instalen en un espacio aislado.
- **Gestión de dependencias**: Todas las dependencias necesarias, como Flask, boto3 y PyMySQL, se listaron en el archivo `requirements.txt`. Esto permite una fácil instalación de las mismas mediante el uso de `pip`.

4.3 Integración con Amazon S3 y RDS

La aplicación web se integra estrechamente con los servicios de AWS **Amazon S3** y **Amazon RDS** para gestionar el almacenamiento de imágenes y la base de datos, respectivamente.

4.3.1 Amazon S3

- **Subida de Imágenes**: La aplicación permite a los usuarios subir imágenes de productos, que se almacenan en el bucket de S3 configurado previamente.
- **Generación de URLs Públicos**: Al subir una imagen, se genera una URL pública que se almacena en la base de datos para mostrar la imagen en el frontend.
- **Gestión de Permisos**: Se aseguraron las políticas de acceso adecuadas para que las imágenes sean accesibles públicamente de manera segura.

4.3.2 Amazon RDS

- **Conexión Segura**: La aplicación utiliza credenciales almacenadas en AWS Secrets Manager para conectarse de manera segura a la base de datos RDS.
- **Operaciones CRUD**: Se implementaron las operaciones de Crear, Leer, Actualizar y Eliminar (CRUD) para gestionar los productos en la base de datos.
- **Manejo de Errores**: Se implementaron mecanismos para manejar posibles errores en las operaciones de base de datos, asegurando la robustez de la aplicación.

4.4 Diseño del Frontend con Bootstrap

Para lograr una interfaz de usuario atractiva y responsiva, se utilizó **Bootstrap**, un framework CSS popular que facilita el diseño de interfaces modernas y adaptables a diferentes dispositivos.

- **Componentes Responsivos**: Se aprovecharon los componentes predefinidos de Bootstrap, como las tarjetas (*cards*), formularios y botones, para construir las páginas de la aplicación.
- **Personalización de Estilos**: Se añadió una hoja de estilos personalizada (`styles.css`) para ajustar y mejorar la apariencia visual según las necesidades específicas del proyecto.
- **Consistencia Visual**: La utilización de Bootstrap garantiza una experiencia de usuario consistente en todas las páginas de la aplicación, mejorando la usabilidad y estética general.

4.5 Funcionalidades Principales

La aplicación web implementa las siguientes funcionalidades clave para gestionar el catálogo de productos:

4.5.1 Agregar Productos

- **Formulario de Registro**: Se proporcionó un formulario intuitivo para que los administradores puedan agregar nuevos productos, incluyendo campos para el nombre, precio, descripción y la imagen del producto.
- **Validación de Datos**: Se implementaron validaciones tanto en el frontend como en el backend para asegurar que los datos ingresados sean correctos y completos.
- **Subida de Imágenes**: Al agregar un producto, la imagen asociada se sube automáticamente al bucket de S3, y su URL se almacena en la base de datos.

4.5.2 Consultar Productos

- **Listado de Productos**: La página principal muestra todos los productos disponibles en la base de datos, presentados de manera organizada y visualmente atractiva.
- **Detalles del Producto**: Al hacer clic en un producto, se puede visualizar información detallada, incluyendo la imagen, descripción y precio.
- **Filtros y Búsqueda**: (Opcional) Se puede implementar funcionalidad de búsqueda y filtrado para mejorar la navegación por el catálogo de productos.

4.5.3 Eliminar Productos

- **Acción de Eliminación**: Se proporciona la opción de eliminar productos directamente desde la interfaz web, con confirmaciones para evitar eliminaciones accidentales.
- **Actualización de la Base de Datos**: Al eliminar un producto, se actualiza la base de datos y, opcionalmente, se puede eliminar la imagen asociada del bucket de S3.

4.6 Configuración Inicial

```
from flask import Flask, render_template, request, redirect, url_for,
    flash
import boto3
import pymysql
import os

app = Flask(__name__)
app.secret_key = 'clave_secreta'

SECRET_NAME = "rds!db-3419db02-547f-45a3-b45e-690fb0626b69"
REGION_NAME = "us-east-1"
BUCKET_NAME = "ecommerce-store-images"
CARPETA_S3 = "products/"
RDS_HOST = "ecommerce-db.cdj5i4puctle.us-east-1.rds.amazonaws.com"
RDS_PORT = 3306
```

Listing 11: Configuración inicial y constantes

4.7 Funciones para Gestión de Secretos y Conexión a la Base de Datos

```
def get_secret():
    session = boto3.session.Session()
    client = session.client(service_name="secretsmanager", region_name=
        REGION_NAME)
    response = client.get_secret_value(SecretId=SECRET_NAME)
    return eval(response['SecretString'])

def get_db_connection(secret, use_db=True):
    return pymysql.connect(
        host=RDS_HOST,
        user=secret["username"],
        password=secret["password"],
        database="ecommerce_db" if use_db else None,
        port=RDS_PORT,
        cursorclass=pymysql.cursors.DictCursor
    )
```

Listing 12: Gestión de secretos y conexión a la base de datos

4.8 Funciones para Configuración de Base de Datos

```python
def create_database_if_not_exists(secret):
    connection = get_db_connection(secret, use_db=False)
    try:
        with connection.cursor() as cursor:
            cursor.execute("CREATE DATABASE IF NOT EXISTS ecommerce_db;")
    finally:
        connection.close()

def create_table_if_not_exists(secret):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("""
                CREATE TABLE IF NOT EXISTS products (
                    id INT AUTO_INCREMENT PRIMARY KEY,
                    name VARCHAR(100) NOT NULL,
                    price DECIMAL(10, 2) NOT NULL,
                    description TEXT,
                    image_url VARCHAR(255) NOT NULL
                );
            """)
            connection.commit()
    finally:
        connection.close()
```

Listing 13: Creación de base de datos y tablas

4.9 Funciones para Gestión de S3 y Productos

```python
def upload_to_s3(file):
    s3 = boto3.client('s3')
    file_name = file.filename
    s3_key = os.path.join(CARPETA_S3, file_name)
    s3.upload_fileobj(file, BUCKET_NAME, s3_key)
    return f"https://{BUCKET_NAME}.s3.{REGION_NAME}.amazonaws.com/{s3_key}"

def fetch_all_products(secret):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("SELECT * FROM products;")
            return cursor.fetchall()
    finally:
        connection.close()

def insert_product_to_db(secret, name, price, description, image_url):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("""
                INSERT INTO products (name, price, description, image_url)
```

```
                VALUES (%s, %s, %s, %s);
            """, (name, price, description, image_url))
            connection.commit()
    finally:
        connection.close()

def delete_product(secret, product_id):
    connection = get_db_connection(secret)
    try:
        with connection.cursor() as cursor:
            cursor.execute("DELETE FROM products WHERE id = %s;", (
                product_id,))
            connection.commit()
    finally:
        connection.close()
```

Listing 14: Subida a S3 y gestión de productos

4.10 Inicialización del Sistema

```
secret = get_secret()
create_database_if_not_exists(secret)
create_table_if_not_exists(secret)
```

Listing 15: Inicialización del sistema

4.11 Definición de Rutas en Flask

```
@app.route('/')
def index():
    products = fetch_all_products(secret)
    return render_template('index.html', products=products)

@app.route('/add', methods=['GET', 'POST'])
def add_product():
    if request.method == 'POST':
        try:
            name = request.form['name']
            price = float(request.form['price'])
            description = request.form['description']
            image = request.files['image']

            if not image:
                flash('La imagen es requerida.', 'danger')
                return redirect(request.url)

            image_url = upload_to_s3(image)
            insert_product_to_db(secret, name, price, description,
                image_url)
            flash(f"Producto '{name}' agregado exitosamente.", 'success
                ')
            return redirect(url_for('index'))
        except Exception as e:
```

```
            flash(f"Error al agregar el producto: {e}", 'danger')
            return redirect(request.url)
    return render_template('add_product.html')

@app.route('/delete/<int:product_id>', methods=['POST'])
def delete_product_route(product_id):
    try:
        delete_product(secret, product_id)
        flash(f"Producto con ID {product_id} eliminado exitosamente.",
            'success')
    except Exception as e:
        flash(f"Error al eliminar el producto: {e}", 'danger')
    return redirect(url_for('index'))
```

Listing 16: Rutas de la aplicación

4.12 Ejecución de la Aplicación

```
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=8000)
```

Listing 17: Ejecución de la aplicación Flask

4.13 Plantilla Base

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Store Ecommerce</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/css/bootstrap.min.css" rel="stylesheet">
    <link rel="stylesheet" href="{{ url_for('static', filename='styles.css') }}">
</head>
<body>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark mb-4">
        <div class="container">
            <a class="navbar-brand" href="{{ url_for('index') }}">Ecommerce Store</a>
            <div class="collapse navbar-collapse">
                <ul class="navbar-nav ms-auto">
                    <li class="nav-item">
                        <a class="nav-link" href="{{ url_for('add_product') }}">Agregar Producto</a>
                    </li>
                </ul>
            </div>
        </div>
    </nav>

    <div class="container">
        {% with messages = get_flashed_messages(with_categories=true) %}
          {% if messages %}
            {% for category, message in messages %}
              <div class="alert alert-{{ category }} alert-dismissible fade show" role="alert">
                {{ message }}
                <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Cerrar"></button>
              </div>
            {% endfor %}
          {% endif %}
        {% endwith %}
        {% block content %}{% endblock %}
    </div>

    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0/dist/js/bootstrap.bundle.min.js"></script>
</body>
</html>
```

Listing 18: Plantilla base para las páginas web

4.14 Página para Agregar Productos

```
{% extends "base.html" %}

{% block content %}
<h1 class="mb-4">Agregar Nuevo Producto</h1>
<form method="POST" enctype="multipart/form-data">
    <div class="mb-3">
        <label for="name" class="form-label">Nombre del Producto</label>
        <input type="text" class="form-control" id="name" name="name" required>
    </div>
    <div class="mb-3">
        <label for="price" class="form-label">Precio</label>
        <input type="number" step="0.01" class="form-control" id="price" name="price" required>
    </div>
    <div class="mb-3">
        <label for="description" class="form-label">Descripci n</label>
        <textarea class="form-control" id="description" name="description" rows="3"></textarea>
    </div>
    <div class="mb-3">
        <label for="image" class="form-label">Imagen del Producto</label>
        <input class="form-control" type="file" id="image" name="image" accept="image/*" required>
    </div>
    <button type="submit" class="btn␣btn-primary">Agregar Producto</button>
</form>
{% endblock %}
```

Listing 19: Plantilla para agregar nuevos productos

4.15 Página de Listado de Productos

```
{% extends "base.html" %}

{% block content %}
<h1 class="mb-4">Productos Disponibles</h1>
{% if products %}
    <div class="row">
        {% for product in products %}
            <div class="col-md-4">
                <div class="card␣mb-4␣shadow-sm">
                    <img src="{{␣product.image_url␣}}" class="card-img-
                       top" alt="{{␣product.name␣}}">
                    <div class="card-body">
                        <h5 class="card-title">{{ product.name }}</h5>
                        <p class="card-text">{{ product.description }}<
                           /p>
                        <div class="d-flex␣justify-content-between␣
                           align-items-center">
                            <span class="text-primary">${{ "%.2f"|
                               format(product.price) }}</span>
                            <form action="{{␣url_for('
                               delete_product_route',␣product_id=
                               product.id)␣}}" method="POST" onsubmit="
                               return␣confirm('␣␣Ests␣␣seguro␣de␣
                               eliminar␣este␣producto?');">
                                <button type="submit" class="btn␣btn-sm
                                   ␣btn-danger">Eliminar</button>
                            </form>
                        </div>
                    </div>
                </div>
            </div>
        {% endfor %}
    </div>
{% else %}
    <p>No hay productos disponibles.</p>
{% endif %}
{% endblock %}
```

Listing 20: Plantilla para mostrar productos disponibles

```
(myenv) C:\Users\Cybor\Desktop\aws>python ecommerce_app\app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployme
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 219-505-238
127.0.0.1 - - [27/Nov/2024 18:21:11] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [27/Nov/2024 18:21:11] "GET /static/styles.css HTTP/1.1" 200 -
127.0.0.1 - - [27/Nov/2024 18:21:11] "GET /favicon.ico HTTP/1.1" 404 -
127.0.0.1 - - [27/Nov/2024 18:22:03] "GET /add HTTP/1.1" 200 -
127.0.0.1 - - [27/Nov/2024 18:22:03] "GET /static/styles.css HTTP/1.1" 304 -
127.0.0.1 - - [27/Nov/2024 18:22:27] "POST /add HTTP/1.1" 302 -
127.0.0.1 - - [27/Nov/2024 18:22:27] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [27/Nov/2024 18:22:27] "GET /static/styles.css HTTP/1.1" 304 -
```

5 Configuración de EC2

La infraestructura del proyecto incluye una instancia de Amazon EC2 configurada para alojar la aplicación de comercio electrónico. A continuación, se detallan las configuraciones empleadas:

5.1 Especificaciones de la Instancia

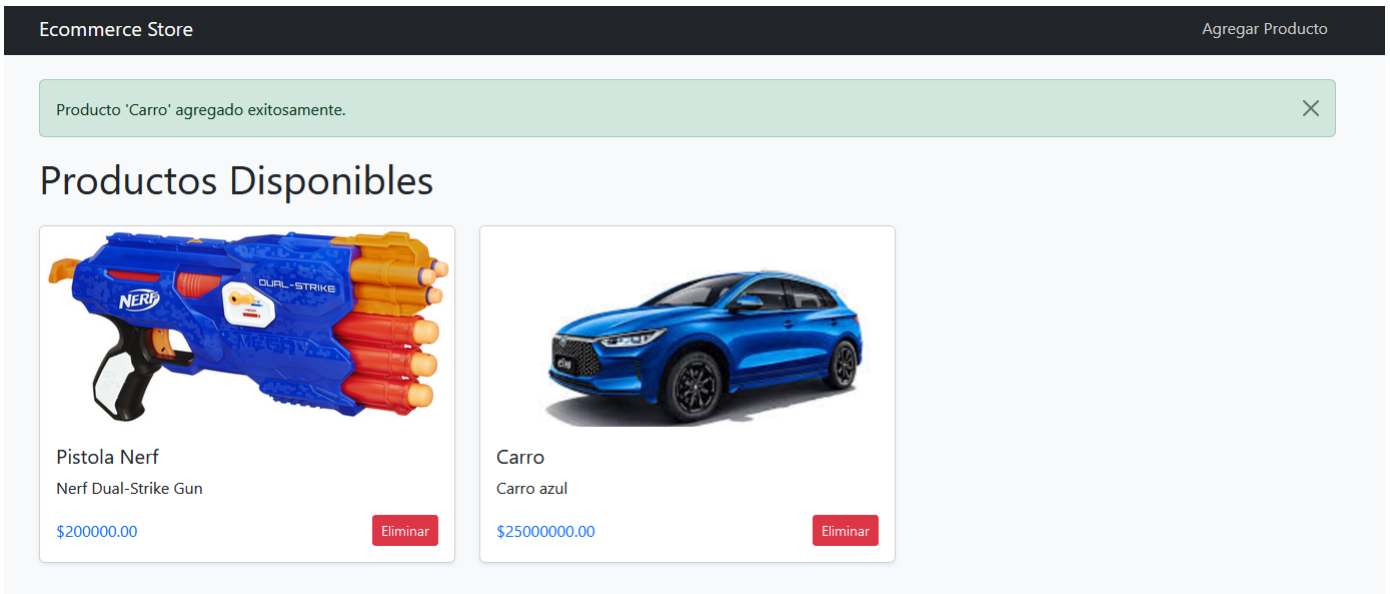
- **Imagen de Amazon Linux**: Se utilizó una imagen de Amazon Linux para la instancia.
- **Grupo de Seguridad**: Configurado para permitir acceso SSH (puerto 22) y tráfico HTTP en el puerto 8000 para la aplicación Flask.
- **Rol IAM**: Asociado con el rol `labuser`, que permite acceso controlado a los servicios de AWS necesarios.
- **Tamaño de la Instancia**: Tipo `t2.micro`, adecuado para aplicaciones ligeras en la capa gratuita.
- **Almacenamiento**: Configuración predeterminada de 8 GB en volúmenes EBS.

5.2 Script de Configuración (User Data)

Para la configuración automática de la instancia, se utilizó el siguiente `user data` script:

```
#!/bin/bash

exec > /var/log/user-data.log 2>&1
set -e
set -x

APP_REPO="https://github.com/caniza1es/aws-ecommerce.git"
APP_DIR="/home/ec2-user/aws-ecommerce"
VENV_DIR="$APP_DIR/myenv"

yum update -y
yum install -y git python3 python3-pip

cd /home/ec2-user
rm -rf aws-ecommerce
git clone $APP_REPO aws-ecommerce
chown -R ec2-user:ec2-user aws-ecommerce

cd $APP_DIR
python3 -m venv $VENV_DIR
source $VENV_DIR/bin/activate
pip install --upgrade pip
pip install -r requirements.txt
deactivate

nohup $VENV_DIR/bin/python3 ecommerce_app/app.py > app.log 2>&1 &
```

Listing 21: Script User Data



6 Configuración de Target Group y Load Balancer

Para garantizar la alta disponibilidad y la distribución eficiente del tráfico hacia la aplicación, se configuró un **Load Balancer** de tipo Application Load Balancer (ALB) en conjunto con un **Target Group** asociado a la instancia de Amazon EC2. A continuación, se describen las configuraciones realizadas.

6.1 Target Group

El Target Group agrupa las instancias o recursos que recibirán tráfico desde el Load Balancer.

6.1.1 Especificaciones del Target Group

- **Nombre del Target Group**: `ecommerce-target`.
- **Tipo de Target**: Instancia (EC2).
- **Protocolo y Puerto**: HTTP en el puerto 8000.
- **VPC**: `vpc-0398778a58ebd53d4`.
- **Verificación de Salud (*Health Check*)**:
  - **Protocolo**: HTTP.
  - **Path**: `/` (Ruta raíz de la aplicación).
  - **Intervalo de Verificación**: 30 segundos.
  - **Cantidad de Intentos Fallidos Permitidos**: 2.

6.1.2 Asociación del Target Group

Se registró la instancia EC2 en el Target Group mediante su ID, asegurando que esté configurada para recibir tráfico en el puerto 8000. Se verificó que la instancia esté en el mismo VPC que el Load Balancer.

6.2 Load Balancer

El Application Load Balancer se configuró para distribuir el tráfico entrante hacia las instancias registradas en el Target Group.

6.2.1 Especificaciones del Load Balancer

- **Tipo de Load Balancer**: Application Load Balancer.
- **Esquema**: Internet-facing.
- **Protocolo de Listener**: HTTPS.
- **Puerto del Listener**: 8000.
- **Zonas de Disponibilidad**:
  - **subnet-0b7254d4acd946344** (us-east-1a).
  - **subnet-06b65501bf51266c7** (us-east-1b).
- **Seguridad**:
  - Asociado al `Security Group` que permite tráfico en el puerto 8000 desde cualquier IP.
  - Certificado SSL/TLS generado e importado para asegurar el tráfico HTTPS.

6.2.2 Configuración del Listener

El Listener del Load Balancer escucha tráfico HTTPS en el puerto 8000 y redirige las solicitudes hacia el Target Group asociado.

Detalles del Listener:

- **Regla por Defecto**: Reenvía todas las solicitudes hacia el Target Group `ecommerce-target`.
- **Política de Seguridad**: Usar un certificado SSL válido para establecer una conexión segura.

6.2.3 Pruebas de Funcionamiento

Para validar la configuración, se accedió al DNS del Load Balancer:

`http://ecommerce-loadbalancer-608229607.us-east-1.elb.amazonaws.com:8000/`

http://ecommerce-loadbalancer-608229607.us-east-1.elb.amazonaws.com:8000/

Ecommerce Store

Productos Disponibles

Pistola Nerf

Nerf Dual-Strike Gun

$200000.00 Eliminar

Carro

Carro azul

$25000000.00 Eliminar

ecommerce-loadbalancer-608229607.us-east-1.elb.amazonaws.com:8000/add

Ecommerce Store

Agregar Nuevo Producto

Nombre del Producto

Bicicleta

Precio

300000

Descripción

Profit dallas fundas internas,acero

Imagen del Producto

Browse... vike.jpg

Agregar Producto

ec2-3-81-227-115.compute-1.amazonaws.com:8000

Ecommerce Store Agregar Producto

Producto 'Bicicleta' agregado exitosamente.

Productos Disponibles

Pistola Nerf

Nerf Dual-Strike Gun

$200000.00 Eliminar

Carro

Carro azul

$25000000.00 Eliminar

Bicicleta

Profit dallas fundas internas,acero

$300000.00 Eliminar

7 Auto Scaling y CloudWatch

Para optimizar la disponibilidad y escalabilidad de la aplicación, se implementaron políticas de **Auto Scaling** y monitoreo continuo mediante **CloudWatch**. A continuación, se describen las configuraciones realizadas.

7.1 Auto Scaling

El grupo de Auto Scaling garantiza que la infraestructura pueda manejar variaciones en la carga de tráfico, escalando dinámicamente la cantidad de instancias EC2 según sea necesario.

7.1.1 Configuración del Grupo de Auto Scaling

- **Nombre del Grupo**: `ecommerce-autoscaling-group`.
- **Tamaño Inicial**: 1 instancia EC2.
- **Tamaño Mínimo**: 1 instancia EC2.
- **Tamaño Máximo**: 3 instancias EC2.
- **Política de Escalado**: Basada en la utilización de CPU.

7.1.2 Políticas de Escalado

Se definieron políticas para manejar la escalabilidad hacia arriba y hacia abajo:

- **Escalar hacia arriba**: Si la utilización promedio de CPU excede el 75 % durante 5 minutos consecutivos.
- **Escalar hacia abajo**: Si la utilización promedio de CPU cae por debajo del 25 % durante 5 minutos consecutivos.

7.1.3 Zonas de Disponibilidad

El grupo de Auto Scaling distribuye instancias en múltiples Zonas de Disponibilidad:

- **subnet-0b7254d4acd946344** (us-east-1a).
- **subnet-06b65501bf51266c7** (us-east-1b).

7.2 Monitoreo con CloudWatch

Se configuraron métricas y alarmas en **Amazon CloudWatch** para monitorear el rendimiento del sistema y garantizar su estabilidad.

7.2.1 Métricas Monitoreadas

- **Utilización de CPU**: Para identificar picos de carga en las instancias EC2.
- **Estado de Salud**: Para verificar que las instancias EC2 estén disponibles y funcionando correctamente.
- **Latencia del Load Balancer**: Para garantizar tiempos de respuesta aceptables.
- **Tráfico de Red**: Para evaluar el volumen de datos enviados y recibidos.

7.2.2 Alarmas Configuradas

- **CPU Utilization High**:
  - **Condición**: Utilización de CPU mayor al 75 %.
  - **Acción**: Escalar hacia arriba (agregar una instancia EC2).
- **CPU Utilization Low**:
  - **Condición**: Utilización de CPU menor al 25 %.
  - **Acción**: Escalar hacia abajo (eliminar una instancia EC2).
- **Target Group Unhealthy Instances**:
  - **Condición**: Instancias en estado no saludable en el Target Group.
  - **Acción**: Notificar mediante SNS (Simple Notification Service).

7.3 Beneficios de Auto Scaling y CloudWatch

- **Escalabilidad Dinámica**: El sistema se adapta automáticamente a los cambios en la demanda.
- **Alta Disponibilidad**: Las instancias se distribuyen en múltiples Zonas de Disponibilidad, reduciendo el impacto de fallas en una única zona.
- **Monitoreo Continuo**: CloudWatch proporciona visibilidad en tiempo real del rendimiento del sistema.

- **Costo-Efectividad**: Se escalan los recursos hacia abajo durante periodos de baja demanda, minimizando los costos operativos.

7.4 Repositorio del Proyecto

El código fuente del sistema de eCommerce está disponible en el siguiente repositorio de GitHub:

`https://github.com/caniza1es/aws-ecommerce`