# EP3. Práctica Programación de aplicaciones con base de datos distribuidas

Ingeniería Tecnologías de la Información 8°A

INTEGRANTES:


- **Mendieta Chimal Sony Luis** **MCSO220598**
- **Ascencio Onofre Carlos Gerardo** **AOCO220155**
- **Nava Sanchez Axel** **NSAO220388**


Docente: LORENZO ANTONIO CARDOSO CONTRERAS

Materia: Tecnologías y aplicaciones en Internet

11/04/2025

Índice

## Introducción.

El presente documento tiene como objetivo describir el desarrollo de un sistema web para la gestión de una perfumería, detallando su propósito, estructura técnica y herramientas utilizadas. El proyecto, titulado "Sistema de Gestión Web para Perfumería", busca optimizar los procesos internos de administración de productos, pedidos y usuarios, ofreciendo una solución intuitiva tanto para el cliente como para el personal administrativo.

Para el almacenamiento de datos se utiliza una base de datos no relacional denominada perfumeria, basada en el modelo de documentos, lo cual permite una estructura flexible y escalable adecuada para el manejo de productos con diversas características. La base de datos se implementa con MongoDB, una de las soluciones NoSQL más populares y eficientes para este tipo de aplicaciones.

El sistema se ejecuta sobre un servidor web orientado a servicios de comercio electrónico, especialmente diseñado para cubrir las necesidades de una tienda de perfumería, permitiendo desde la navegación y búsqueda de productos hasta la gestión de compras, pedidos y usuarios con distintos niveles de acceso.

Este documento ofrece un recorrido por la arquitectura y componentes principales del sistema, enfocándose en la solución técnica adoptada, la problemática abordada, y el manejo eficiente de la base de datos no relacional en un entorno web moderno.

## Planteamiento del problema.

Actualmente, la perfumería enfrenta limitaciones significativas en cuanto a su alcance comercial, la gestión eficiente de sus operaciones internas y la administración general del negocio. El modelo tradicional, basado principalmente en la atención presencial y el registro manual de productos, pedidos y clientes, dificulta el crecimiento del negocio y reduce su competitividad frente a otras empresas que ya han adoptado soluciones digitales.

Uno de los principales problemas identificados es la falta de una plataforma web que permita ampliar la búsqueda y captación de nuevos clientes a través de medios digitales. Esta carencia limita la visibilidad de los productos, restringe las oportunidades de venta y no permite aprovechar las ventajas del comercio electrónico.

Además, el proceso actual de administración y control de inventario, pedidos y usuarios se realiza de forma poco automatizada, lo que genera duplicidad de tareas, errores humanos y pérdida de tiempo. Esto afecta directamente la eficiencia operativa del negocio, dificultando el seguimiento de productos, el análisis de ventas y la atención personalizada al cliente.

La situación ideal a la que se desea llegar con el desarrollo de este sistema es la implementación de una plataforma web moderna, que funcione como un canal de ventas en línea, facilite la gestión de productos y pedidos, y permita una administración centralizada con distintos niveles de acceso según el tipo de usuario. Con esta solución, se espera:

- Ampliar el alcance del negocio a nuevos clientes mediante presencia online.
- Automatizar procesos internos para mejorar la eficiencia y reducir errores.
- Incrementar las ventas mediante una experiencia de compra más accesible y cómoda.
- Facilitar la gestión administrativa mediante un panel intuitivo y estructurado.

En conclusión, la aplicación web busca solventar las principales limitaciones operativas de la perfumería, transformándola en un negocio más competitivo, ágil y preparado para enfrentar los retos del entorno digital actual.

## Base de datos no Relacional.

Para el desarrollo del sistema web de la perfumería se ha utilizado MongoDB, una base de datos NoSQL de tipo documento. Este tipo de base de datos permite almacenar los datos en formato JSON (o BSON internamente), facilitando la representación flexible de la información y adaptándose perfectamente a las necesidades del sistema, donde los datos pueden tener estructuras dinámicas.

Se ha diseñado una base de datos llamada perfumeria, la cual contiene varias colecciones que representan las entidades principales del sistema. Estas colecciones no están relacionadas de forma rígida como en una base de datos relacional, lo que permite escalar y modificar su estructura con mayor facilidad.

A continuación, se muestra el esquema físico con ejemplos de documentos de dos de las colecciones principales:

En la siguiente figura se muestra la colección de producto.

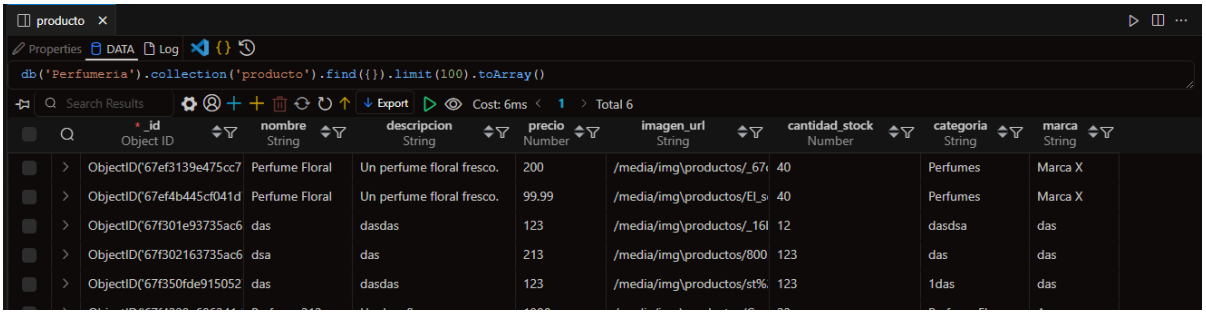



***Figura 1.1 Colección Producto.***

En la siguiente figura se muestra la colección de proveedor.

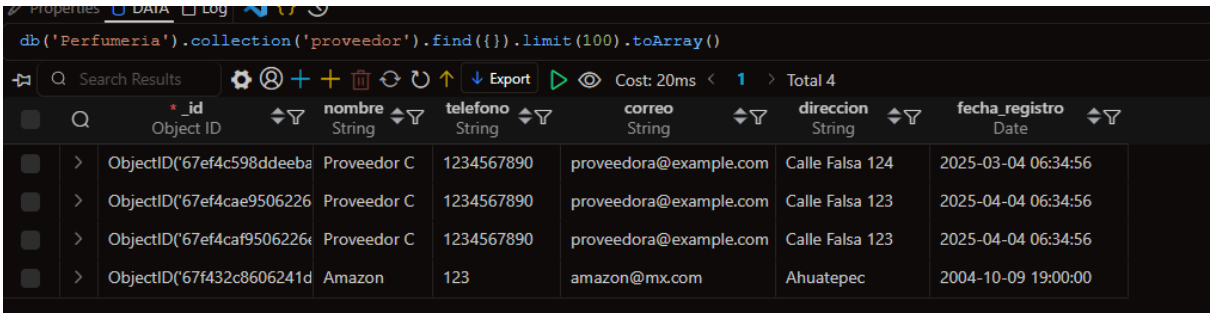



***Figura 1.2 Colección Producto.***

## Manejo de sesiones.

El sistema web de la perfumería implementa un sistema de autenticación basado en sesiones para gestionar el acceso de los usuarios y sus permisos dentro de la plataforma. A continuación, se describen los diferentes tipos de usuarios y sus permisos, así como el uso de tokens para la API.

1. Administrador

- Permisos: El Administrador tiene acceso completo a todas las funcionalidades del sistema. Puede modificar los productos, proveedores, y usuarios, así como realizar otras operaciones administrativas.

- Acceso: Solo el Administrador puede acceder a rutas que permiten la modificación de datos, configuración del sistema y gestión de otros usuarios.

- Acción: El Administrador tiene la capacidad de modificar los productos, proveedores, y usuarios, además de gestionar configuraciones del sistema.

- API y Tokens: El Administrador es el único usuario que podrá obtener y utilizar un token de autenticación para interactuar con la API. El token se utiliza para validar las solicitudes a las rutas protegidas de la API y garantizar que solo el Administrador pueda realizar acciones de gestión (como modificar productos o proveedores).

2. Empleado

- Permisos: Los Empleados tienen permisos limitados y solo pueden editar su propio perfil, sin acceso a la gestión de datos del sistema ni funciones administrativas.

- Acceso: Los Empleados solo tienen acceso a las rutas que les permiten actualizar su información personal (por ejemplo, nombre, correo, etc.).

- Acción: Los Empleados pueden editar su perfil, pero no tienen acceso a ninguna función de administración o modificación de datos en el sistema.

- API y Tokens: Los Empleados no tendrán acceso a la API para realizar modificaciones de datos. Solo pueden interactuar con las funcionalidades básicas que no requieren autenticación vía token.

3. Usuarios no autenticados

- Permisos: Los Usuarios que no han iniciado sesión no podrán acceder a las funcionalidades protegidas del sistema.

- Acceso: Este grupo de usuarios solo podrá visualizar el contenido público del sitio (como el catálogo de productos), pero no tendrá acceso a ninguna funcionalidad restringida (como la edición de perfil o la gestión de datos).

- Acción: No podrán realizar ninguna acción de edición o gestión hasta que inicien sesión y obtengan los permisos correspondientes.

- API y Tokens: Los Usuarios no autenticados no tienen acceso a la API ni a las rutas protegidas por token. Solo podrán acceder a las funcionalidades públicas del sistema.

Sistema de autenticación y autorización

- Autenticación por sesión: El sistema utiliza un mecanismo de sesión para autenticar a los usuarios en la plataforma web. Las credenciales de los usuarios se validan al iniciar sesión, y la sesión se mantiene activa durante su interacción con el sistema.

- Tokens para la API: Para las interacciones con la API, se implementa un sistema de autenticación por token. El Administrador es el único que obtiene un token JWT (JSON Web Token) al iniciar sesión. Este token se incluye en las solicitudes a la API para validar que el Administrador tiene permisos para realizar acciones de gestión (como agregar o modificar productos o proveedores).

    - El token es generado en el momento del login y tiene una duración determinada. El Administrador debe incluir el token en las cabeceras de las solicitudes API, lo que permitirá al servidor verificar su identidad y los permisos asociados.

- Autorización basada en roles: El sistema de autorización se encarga de verificar el rol de cada usuario para asegurar que solo el Administrador tenga acceso completo a las rutas y funcionalidades protegidas. Los Empleados tienen permisos limitados, mientras que los Usuarios no autenticados tienen acceso solo a lo que es público.

## Código fuente.

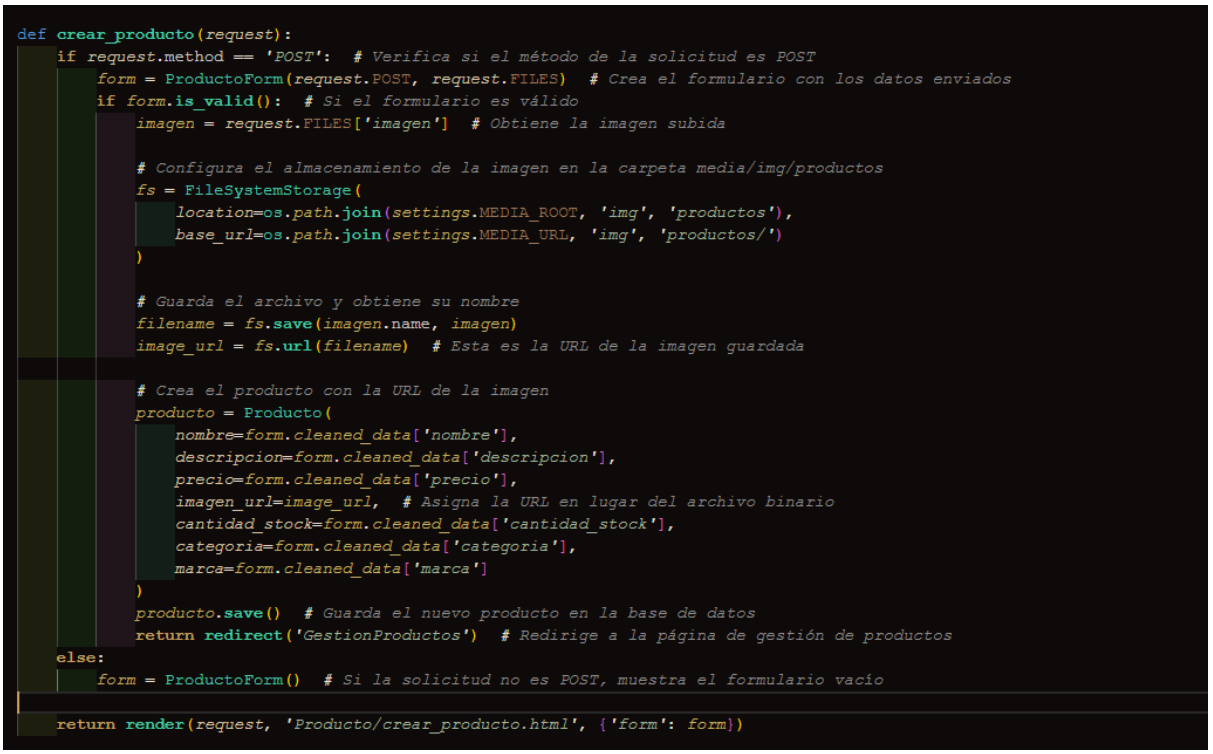
Descripción:

Este fragmento de código (figura 1.3) maneja la lógica para mostrar la página de inicio del administrador. Verifica si el usuario que está intentando acceder es un administrador. Si no lo es, lo redirige a la página de cierre de sesión. Además, obtiene estadísticas relevantes como el número de productos, proveedores y usuarios para mostrarlas en el panel de administración.

```python
from django.shortcuts import redirect
from functools import wraps



def Inicio_admin(request):
    # Obtener el ID del administrador desde la sesión
    admin_id = request.session.get('usuario_id')  # Se obtiene el ID del usuario actual desde la sesión activa

    try:
        # Intentar obtener el objeto de administrador basado en el ID de la sesión
        admin = Usuario.objects.get(id=admin_id, tipo='admin')  # Se busca el usuario con el ID y tipo 'admin'
    except Usuario.DoesNotExist:
        # Si no se encuentra el administrador con ese ID, redirige a la página de cierre de sesión
        return redirect('cerrar_sesion')  # Si el administrador no existe o no es un administrador, se cierra la sesión

    # Obtener estadísticas para mostrar en la vista del panel de administrador
    context = {
        'admin': admin,  # Se pasa el objeto del administrador para usarlo en la plantilla
        'total_productos': Producto.objects.count(),  # Contar el total de productos en la base de datos
        'total_proveedores': Proveedor.objects.count(),  # Contar el total de proveedores en la base de datos
        'total_usuarios': Usuario.objects.count()  # Contar el total de usuarios en la base de datos
    }

    # Renderizar la vista del administrador con el contexto de los datos
    return render(request, 'Admin/Inicio_admin.html', context)  # Renderiza la plantilla 'Inicio_admin.html' pasando el contexto con los datos
```

***Figura 1.3 Inicio Administrador***

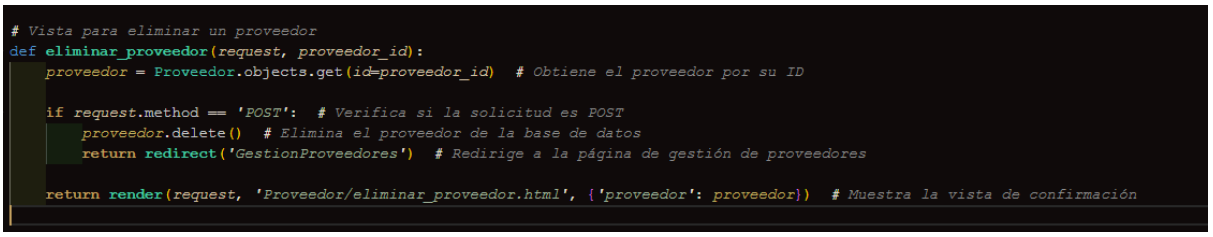
Este fragmento de código (figura 1.4) maneja la creación de un nuevo producto en el sistema. Permite al administrador subir una imagen asociada al producto y guarda esta imagen en una carpeta específica dentro del directorio media/img/productos. La URL de la imagen es almacenada en la base de datos, mientras que los datos del producto (como nombre, descripción, precio, cantidad de stock, etc.) se guardan en la base de datos. Si el formulario es válido, el producto es creado y se redirige a la página de gestión de productos.



```python
def crear_producto(request):
    if request.method == 'POST':  # Verifica si el método de la solicitud es POST
        form = ProductoForm(request.POST, request.FILES)  # Crea el formulario con los datos enviados
        if form.is_valid():  # Si el formulario es válido
            imagen = request.FILES['imagen']  # Obtiene la imagen subida

            # Configura el almacenamiento de la imagen en la carpeta media/img/productos
            fs = FileSystemStorage(
                location=os.path.join(settings.MEDIA_ROOT, 'img', 'productos'),
                base_url=os.path.join(settings.MEDIA_URL, 'img', 'productos/')
            )

            # Guarda el archivo y obtiene su nombre
            filename = fs.save(imagen.name, imagen)
            image_url = fs.url(filename)  # Esta es la URL de la imagen guardada

            # Crea el producto con la URL de la imagen
            producto = Producto(
                nombre=form.cleaned_data['nombre'],
                descripcion=form.cleaned_data['descripcion'],
                precio=form.cleaned_data['precio'],
                imagen_url=image_url,  # Asigna la URL en lugar del archivo binario
                cantidad_stock=form.cleaned_data['cantidad_stock'],
                categoria=form.cleaned_data['categoria'],
                marca=form.cleaned_data['marca']
            )
            producto.save()  # Guarda el nuevo producto en la base de datos
            return redirect('GestionProductos')  # Redirige a la página de gestión de productos
    else:
        form = ProductoForm()  # Si la solicitud no es POST, muestra el formulario vacío

    return render(request, 'Producto/crear_producto.html', {'form': form})
```

***Figura 1.4 Crear producto.***

Este fragmento de código (figura 1.5) maneja la eliminación de un proveedor del sistema. Permite al administrador eliminar un proveedor específico, después de confirmar la acción en una vista que muestra los detalles del proveedor. Si el método de solicitud es POST, el proveedor es eliminado de la base de datos. Luego, se redirige al administrador a la página de gestión de proveedores.



```python
# Vista para eliminar un proveedor
def eliminar_proveedor(request, proveedor_id):
    proveedor = Proveedor.objects.get(id=proveedor_id)  # Obtiene el proveedor por su ID

    if request.method == 'POST':  # Verifica si la solicitud es POST
        proveedor.delete()  # Elimina el proveedor de la base de datos
        return redirect('GestionProveedores')  # Redirige a la página de gestión de proveedores

    return render(request, 'Proveedor/eliminar_proveedor.html', {'proveedor': proveedor})  # Muestra la vista de confirmación
```

***Figura 1.5 Eliminar Proveedor.***

## Herramientas middleware

El LoginSerializer es utilizado para autenticar a los usuarios en el sistema. Específicamente, se encarga de verificar las credenciales de inicio de sesión, validando el correo y la contraseña proporcionados por el usuario. El middleware maneja la validación de las credenciales y la verificación del tipo de usuario (en este caso, se valida que sea un administrador) asi como se muestra en la figura 1.6

```python
class LoginSerializer(serializers.Serializer):
    correo = serializers.EmailField()
    contraseña = serializers.CharField(write_only=True)

    def validate(self, data):
        correo = data.get("correo")
        contraseña = data.get("contraseña")

        try:
            usuario = Usuario.objects.get(correo=correo)
        except Usuario.DoesNotExist:
            raise serializers.ValidationError("Usuario no encontrado")

        # Verifica la contraseña (¡debe estar hasheada en la base de datos!)
        if not check_password(contraseña, usuario.contraseña):
            raise serializers.ValidationError("Contraseña incorrecta")

        if usuario.tipo != 'admin':  # Asegúrate de que el valor que defines en 'tipo' sea 'ad
            raise serializers.ValidationError("No tienes permisos de administrador")

        return usuario
```

***Figura 1.7 Serialización para conseguir el token.***

El ProductoSerializer (figura 1.8) se utiliza para manejar las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) de productos en la base de datos. Esta herramienta transforma los objetos de productos en un formato JSON que puede ser utilizado por el front-end o por otras aplicaciones. También define cómo se deben crear y actualizar los productos en la base de datos de MongoDB.

```python
class ProductoSerializer(serializers.Serializer):
    id = serializers.CharField(read_only=True)  # ObjectId como cadena
    nombre = serializers.CharField()
    precio = serializers.DecimalField(max_digits=10, decimal_places=2)
    descripcion = serializers.CharField()
    imagen_url = serializers.CharField()
    cantidad_stock = serializers.IntegerField()
    categoria = serializers.CharField()
    marca = serializers.CharField()

    def create(self, validated_data):
        return Producto.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.nombre = validated_data.get('nombre', instance.nombre)
        instance.precio = validated_data.get('precio', instance.precio)
        instance.descripcion = validated_data.get('descripcion', instance.descripcion)
        instance.imagen_url = validated_data.get('imagen_url', instance.imagen_url)
        instance.cantidad_stock = validated_data.get('cantidad_stock', instance.cantidad_stock)
        instance.categoria = validated_data.get('categoria', instance.categoria)
        instance.marca = validated_data.get('marca', instance.marca)
        instance.save()
        return instance
```

***Figura 1.8 Serialización Producto.***

El ProveedorSerializer (figua 1.9) se utiliza para realizar operaciones CRUD con los proveedores. Al igual que el ProductoSerializer, este se encarga de convertir los datos del proveedor en un formato adecuado para su transmisión a través de la API, y maneja las operaciones de creación y actualización de proveedores.

```python
class ProveedorSerializer(serializers.Serializer):
    id = serializers.CharField(read_only=True)  # ObjectId como cadena
    nombre = serializers.CharField(max_length=255)
    telefono = serializers.CharField(max_length=15)
    correo = serializers.EmailField()
    direccion = serializers.CharField(max_length=255, allow_null=True, allow_blank=True)
    fecha_registro = serializers.DateTimeField()  # Ahora es editable

    def create(self, validated_data):
        return Proveedor.objects.create(**validated_data)

    def update(self, instance, validated_data):
        instance.nombre = validated_data.get('nombre', instance.nombre)
        instance.telefono = validated_data.get('telefono', instance.telefono)
        instance.correo = validated_data.get('correo', instance.correo)
        instance.direccion = validated_data.get('direccion', instance.direccion)
        instance.fecha_registro = validated_data.get('fecha_registro', instance.fecha_registro)
        instance.save()
        return instance
```

***Figura 1.9 Serialización Proveedor***

El MongoEngineJWTAuthentication (figura 2.0) es un middleware personalizado que se utiliza para autenticar a los usuarios a través de JSON Web Tokens (JWT). El sistema valida que el token enviado en la cabecera de la solicitud sea válido y corresponde a un usuario registrado en la base de datos. Si el token es válido, permite que el usuario acceda a la información. Si el token es inválido o el usuario no existe, se lanzará una excepción y no se permitirá el acceso.

```python
# authentication.py
from rest_framework.authentication import BaseAuthentication
from rest_framework.exceptions import AuthenticationFailed
from rest_framework_simplejwt.tokens import UntypedToken
from rest_framework_simplejwt.settings import api_settings
from .models import Usuario

class MongoEngineJWTAuthentication(BaseAuthentication):
    def authenticate(self, request):
        header = request.META.get('HTTP_AUTHORIZATION', '')
        if not header.startswith('Bearer '):
            return None

        raw_token = header.split(' ')[1]
        try:
            token = UntypedToken(raw_token)
        except Exception:
            raise AuthenticationFailed('Token inválido')

        user_id = token.get(api_settings.USER_ID_CLAIM)
        try:
            user = Usuario.objects.get(id=user_id)
        except Usuario.DoesNotExist:
            raise AuthenticationFailed('Usuario no encontrado')

        return (user, token)
```

***Figura 2.0 Autenticación para la api (Middleware).***

El ProveedorAPI y el ProveedorDetailAPI (figuras 2.1, 2.2) son vistas de API para gestionar los proveedores en la aplicación. Estas vistas están protegidas por autenticación JWT (JSON Web Token) a través del middleware MongoEngineJWTAuthentication, lo que garantiza que solo los usuarios autenticados puedan acceder a ellas.

Las funcionalidades incluyen:

1. Obtener todos los proveedores (GET /proveedores).
2. Crear un nuevo proveedor (POST /proveedores).
3. Obtener un proveedor específico por ID (GET /proveedores/{id}).
4. Actualizar los datos de un proveedor (PUT /proveedores/{id}).
5. Eliminar un proveedor (DELETE /proveedores/{id}).

```python
class ProveedorAPI(APIView):
    authentication_classes = [MongoEngineJWTAuthentication]  # Usa tu backend
    permission_classes = [IsAuthenticated]
    # Obtener todos los proveedores o crear uno nuevo
    def get(self, request, *args, **kwargs):
        proveedores = Proveedor.objects.all()  # Queryset de MongoEngine
        serializer = ProveedorSerializer(proveedores, many=True)
        return Response(serializer.data)

    def post(self, request, *args, **kwargs):
        serializer = ProveedorSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

***Figura 2.1 API Proveedor***

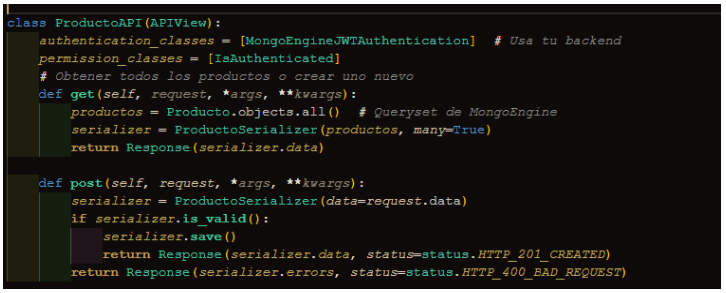
```python
class ProveedorDetailAPI(APIView):
    authentication_classes = [MongoEngineJWTAuthentication]  # Usa tu backend
    permission_classes = [IsAuthenticated]
    # Helper para obtener un proveedor por ID
    def get_object(self, id):
        try:
            return Proveedor.objects.get(id=id)
        except Proveedor.DoesNotExist:
            raise status.HTTP_404_NOT_FOUND

    # Obtener un proveedor específico
    def get(self, request, id, *args, **kwargs):
        proveedor = self.get_object(id)
        serializer = ProveedorSerializer(proveedor)
        return Response(serializer.data)

    # Actualizar un proveedor
    def put(self, request, id, *args, **kwargs):
        proveedor = self.get_object(id)
        serializer = ProveedorSerializer(proveedor, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    # Eliminar un proveedor
    def delete(self, request, id, *args, **kwargs):
        proveedor = self.get_object(id)
        proveedor.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

***Figura 2.2 API Proveedor***

Las clases ProductoAPI y ProductoDetailAPI (figura 2.3, 2.4) son vistas basadas en clases (APIView) que exponen una API RESTful para el manejo de productos. Estas vistas están protegidas por autenticación con JSON Web Tokens (JWT), usando el middleware personalizado MongoEngineJWTAuthentication, garantizando que solo usuarios autenticados (como el administrador) puedan realizar operaciones sobre los productos.

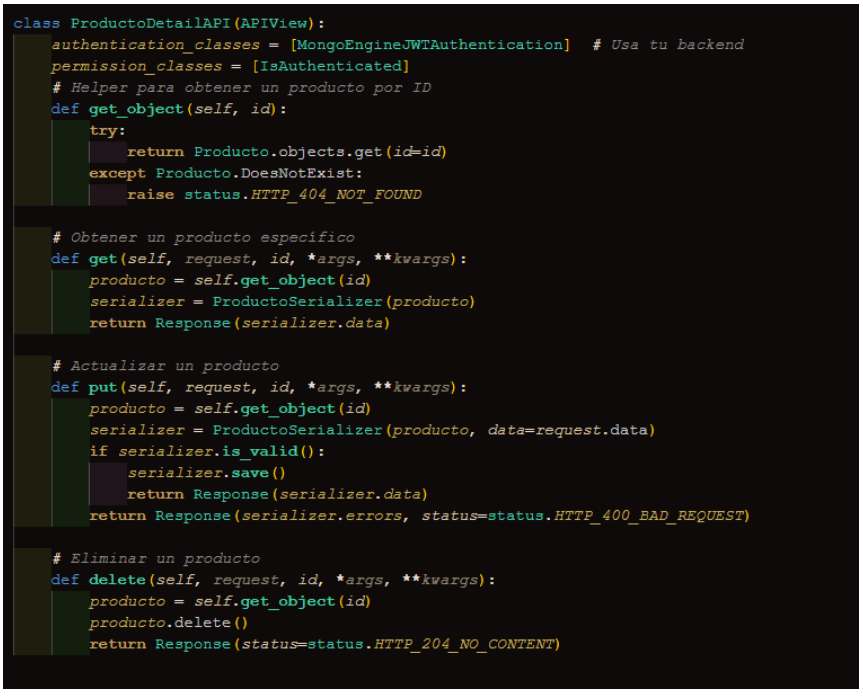
Funcionalidades de la API:

- GET /productos/ → Obtener todos los productos.
- POST /productos/ → Crear un nuevo producto.
- GET /productos/{id} → Obtener un producto por ID.
- PUT /productos/{id} → Actualizar un producto.
- DELETE /productos/{id} → Eliminar un producto.



```python
class ProductoAPI(APIView):
    authentication_classes = [MongoEngineJWTAuthentication]  # Usa tu backend
    permission_classes = [IsAuthenticated]
    # Obtener todos los productos o crear uno nuevo
    def get(self, request, *args, **kwargs):
        productos = Producto.objects.all()  # Queryset de MongoEngine
        serializer = ProductoSerializer(productos, many=True)
        return Response(serializer.data)

    def post(self, request, *args, **kwargs):
        serializer = ProductoSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

***Figura 2.3 API Producto.***



```python
class ProductoDetailAPI(APIView):
    authentication_classes = [MongoEngineJWTAuthentication]  # Usa tu backend
    permission_classes = [IsAuthenticated]
    # Helper para obtener un producto por ID
    def get_object(self, id):
        try:
            return Producto.objects.get(id=id)
        except Producto.DoesNotExist:
            raise status.HTTP_404_NOT_FOUND

    # Obtener un producto específico
    def get(self, request, id, *args, **kwargs):
        producto = self.get_object(id)
        serializer = ProductoSerializer(producto)
        return Response(serializer.data)

    # Actualizar un producto
    def put(self, request, id, *args, **kwargs):
        producto = self.get_object(id)
        serializer = ProductoSerializer(producto, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    # Eliminar un producto
    def delete(self, request, id, *args, **kwargs):
        producto = self.get_object(id)
        producto.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

***Figura 2.4 API Producto.***

## Consultas de la BD no relacional.

Este fragmento (figura 2.5) muestra la implementación del LoginSerializer, utilizado para autenticar a los usuarios por medio del correo y la contraseña. Si el usuario es válido y tiene permisos de administrador, se genera un token JWT que será utilizado para acceder a las APIs protegidas. Esto garantiza un acceso seguro solo a usuarios autorizados.

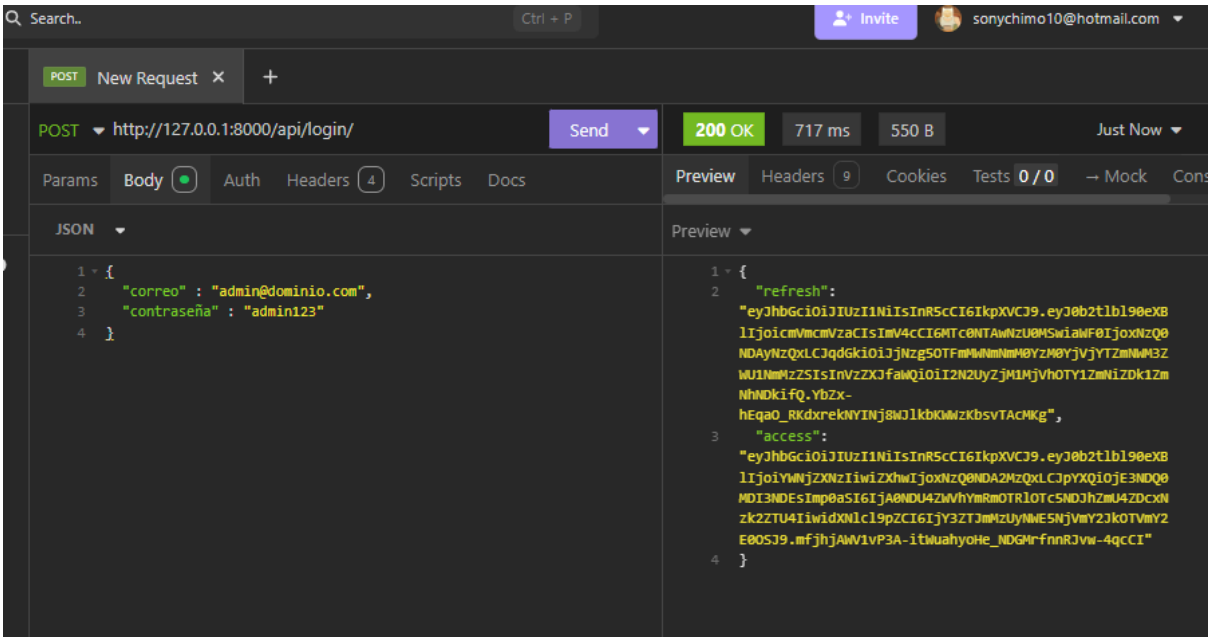



***Figura 2.5 Obtención de token***

Este fragmento (figura 2.6) muestra el uso del método GET en la clase ProductoAPI, el cual permite obtener todos los productos almacenados en la base de datos no relacional. Utiliza MongoEngine para recuperar los datos y ProductoSerializer para convertirlos en formato JSON. Esta funcionalidad permite a los administradores visualizar el inventario completo a través de la API.

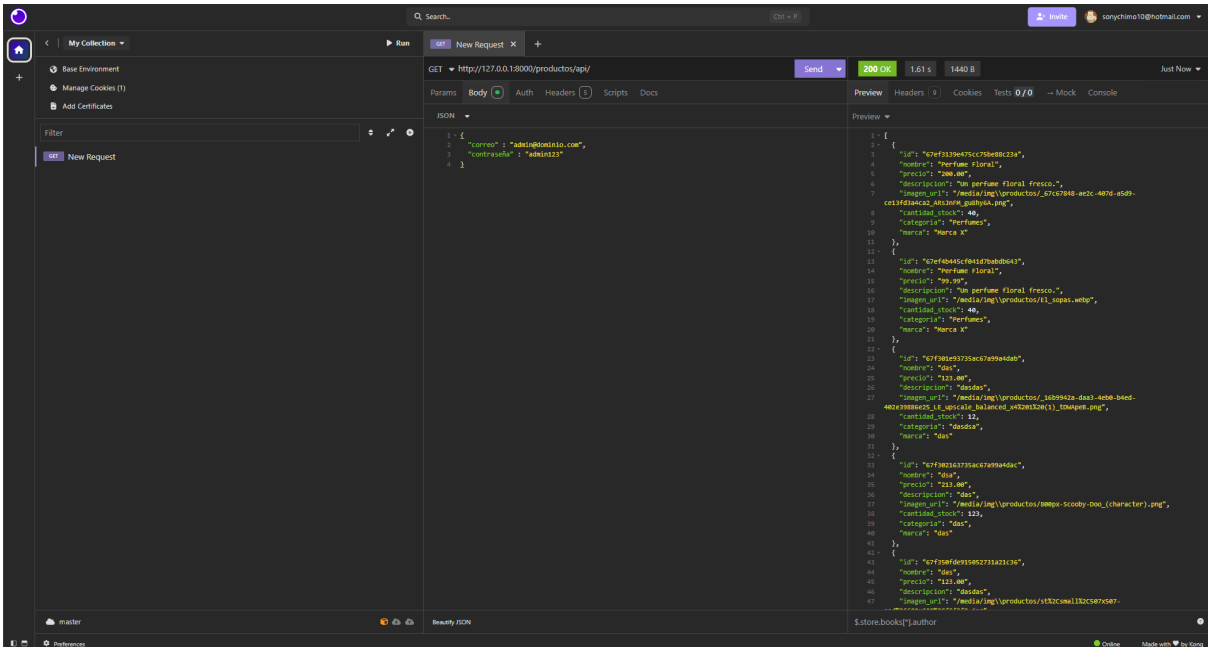



***Figura 2.6 API Producto GET.***

En esta sección (figura 2.7) se muestra el uso del método POST dentro de la clase ProductoAPI, el cual permite registrar nuevos productos en la base de datos no relacional. El administrador envía los datos necesarios (como nombre, precio, descripción, stock, etc.) en formato JSON.

El ProductoSerializer valida la información antes de guardarla en MongoDB a través de MongoEngine. Esta operación está protegida por autenticación mediante JWT, lo que asegura que solo los usuarios con permisos válidos (como el administrador) puedan acceder a esta función.

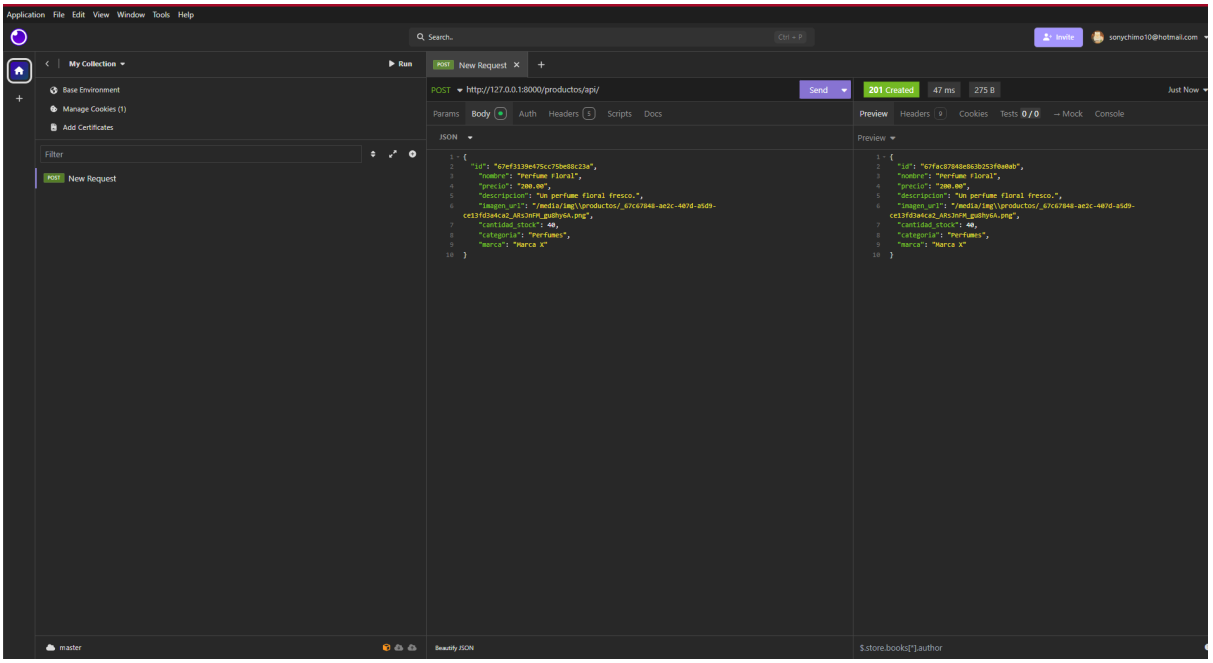



***Figura 2.7 API Producto POST***

Esta captura (figura 2.8) muestra el uso del método PUT en la clase ProductoDetailAPI, el cual permite actualizar la información de un producto específico mediante su ID.

El administrador puede enviar un cuerpo JSON con los campos a modificar, como nombre, precio, descripción, imagen, entre otros. Antes de aplicar los cambios, los datos se validan mediante el ProductoSerializer.

Este endpoint también está protegido por autenticación JWT personalizada (MongoEngineJWTAuthentication) para asegurar que solo el administrador tenga acceso a modificar registros en la base de datos MongoDB.

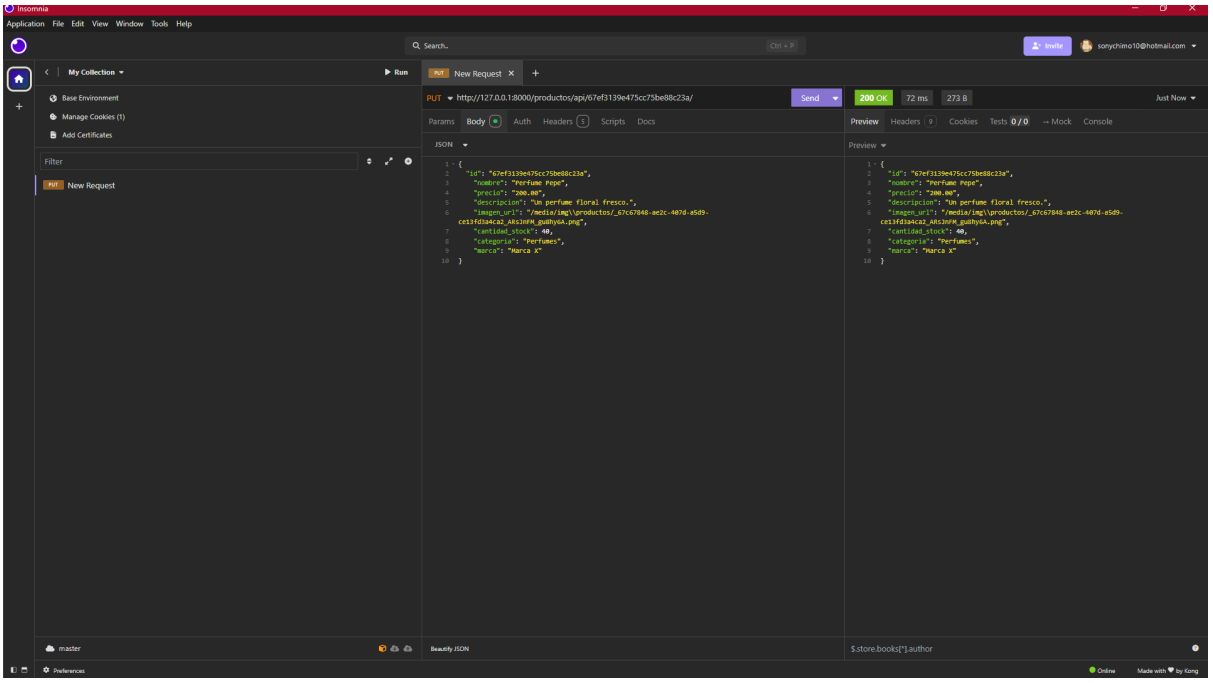



***Figura 2.8 API Producto PUT***

## Conclusiones.

Mendieta Chimal Sony Luis:
La implementación de MongoDB como base de datos NoSQL demostró ser clave para manejar estructuras flexibles de datos, como productos y proveedores con atributos dinámicos. Esto permitió adaptarse rápidamente a cambios en los requisitos del sistema sin necesidad de modificar esquemas rígidos, optimizando el desarrollo del sistema de gestión para la perfumería.

Ascencio Onofre Carlos Gerardo:
Uno de los desafíos más significativos fue garantizar la seguridad y escalabilidad del sistema, especialmente al integrar autenticación basada en roles con tokens JWT y middleware personalizado. La configuración de permisos diferenciados (administrador, empleado y usuarios no autenticados) requirió un manejo cuidadoso de las sesiones y validación de tokens para evitar vulnerabilidades.

Nava Sanchez Axel:
Durante el proceso, destacó la importancia de herramientas como los serializadores (ProductoSerializer, ProveedorSerializer) para transformar datos entre MongoDB y la API RESTful. Además, la integración exitosa de consultas NoSQL con operaciones CRUD mediante MongoEngine facilitó la interacción eficiente con la base de datos, aunque inicialmente se enfrentaron errores de sintaxis y configuración en las consultas.