Gestión de API dentro de ecommerce web ‘MXDiscovery”

Felix Verdugo J.J, Cruz Arredondo José D., Sánchez Cuevas Fernanda, Humarán Beltrán Saul A.

INTRODUCCIÓN

Por medio del siguiente documento se explicará la utilidad de cada API utilizada dentro de las 3 aplicaciones encontradas en nuestro entorno de desarrollo, estos mismos serán acompañados por fragmentos de código para señalar exactamente a que se refiere cada explicación, en que archivo ‘.py’ se encuentra y en que app está importada dicha API.
De esta misma forma, se presentarán los serializers de cada aplicación explicando la lógica que estos siguen para el correcto funcionamiento de la pagina.

I.- Aplicación de Usuarios (users).

A.- Base de datos; Creación de tabla(s) (Models.py).

```python
class User(AbstractBaseUser, PermissionsMixin):
    email = models.CharField(max_length=100, unique=True)
    name = models.CharField(max_length=100)
    last_name = models.CharField(max_length=100)
    avatar = models.ImageField(default="avatar.png")
    date_joined = models.DateTimeField(default=timezone.now)
    is_staff = models.BooleanField(default=False)
    objects = CustomUserManager()
    USERNAME_FIELD = "email"
    REQUIRED_FIELDS = []
```

Dentro de la creación de la tabla se forman los valores que son los que distinguen a cada usuario. Aquí una pequeña explicación de lo cada valor siginifica:

- Email: Objeto tipo texto (string) el cual acepta un límite de caracteres máximo de 100 y con valor único para evitar la creación múltiple de cuentas con un solo email.
- Name/Last Name: Objeto tipo texto (string) donde almacena el nombre y apellidos de un usuario
- Avatar: Objeto de imagen donde el usuario puede subir su foto de perfil. En caso de no tener ninguno se guardará uno por defecto.
- Date Joined: Valor de tipo fecha donde se usa la zona horaria del usuario para almacenar la fecha de creación de su cuenta.
- IsStaff: Valor booleano donde se determina el nivel de acceso del usuario.

Con ayuda de esta base de datos es que la API funciona por lo que es importante tener una breve explicación de cada objeto y/o valor que la base de datos incluye.

B.- Conexión Backend/Frontend (Serializers.py)

```python
class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["email", "name", "last_name", "id", "avatar"]


class RegisterUserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ["email", "name", "last_name", "password"]


class MyTokenObtainPairSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)

        token['email'] = user.email
        token['avatar'] = user.avatar.url
        token['is_staff'] = user.is_staff
        token['name'] = user.name
        token['last_name'] = user.last_name
```

Como se puede apreciar, este archivo cuenta con 3 clases las cuales permiten consultar, registrar y actualizar tokens de cada usuario de la web. Este ultimo es posible gracias a la configuración del backend por medio del framework ‘rest’ y su función ‘jwt’, las cuales ayudan a que despues de cierto tiempo de inactividad, se cierre sesión automaticamente con el objetivo de mantener protegido al usuario.
Solo fue importante la importación del archivo *Models.py* , y cada valor señalado dentro de este archivo para el correcto funcionamiento.
La función de este archivo es dar lógica a los *request* dentro del archivo *Views.py*.

C.- Requests/APIs (Views,py):

Esta API permite obtener información detallada de un usuario específico mediante una solicitud GET. La autenticación es requerida, asegurando que solo los usuarios autenticados tengan acceso a esta funcionalidad. El parámetro pk identifica al usuario deseado, y la API recupera los datos asociados a ese usuario desde la base de datos. Posteriormente, utiliza un serializador (UserSerializer) para convertir estos datos en un formato adecuado para ser transmitido como respuesta. Finalmente, la información del usuario se envía como una respuesta en formato de datos serializados.

```python
@api_view(['GET'])
@permission_classes([IsAuthenticated])
def get_solo_user(request, pk):
    user = User.objects.get(pk=pk)
    serializer = UserSerializer(user)
    return Response(serializer.data)
```

API 2: Editar perfil de usuario
Esta API permite la modificación del perfil de un usuario mediante una solicitud PUT. Se busca al usuario asociado al correo electrónico proporcionado en la URL (email). En caso de que el usuario no exista, la API devuelve una respuesta con el código de estado 404 (Not Found).
Si el usuario existe y la solicitud proviene del mismo usuario autenticado, la API utiliza un serializador (UserSerializer) para actualizar los datos del usuario con la información proporcionada en la solicitud. Si la validación del serializador es exitosa, los cambios se guardan y la API responde con los datos actualizados. En caso de errores de validación, la API devuelve una respuesta con el código de estado 400 (Bad Request) y detalles sobre los errores encontrados.
Si la solicitud no proviene del mismo usuario autenticado, se responde con el código de estado 401 (Unauthorized), indicando que el usuario no tiene permisos para realizar la operación de edición en el perfil.

```python
@api_view(['PUT'])
def edit_profile(request, email):
    try:
        user = User.objects.get(email=email)
    except User.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.user == user:
        serializer = UserSerializer(user, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        else:
            return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
    else:
        return Response(status=status.HTTP_401_UNAUTHORIZED)
```

API 3: Búsqueda de usuarios
Esta API permite realizar búsquedas de usuarios en función de un parámetro de consulta. La solicitud GET permite incluir un parámetro query a través de los parámetros de la URL. Si no se proporciona ningún parámetro, la API asume una cadena vacía como valor predeterminado para query.
La API utiliza el parámetro query para realizar una búsqueda en la base de datos de usuarios, filtrando aquellos cuyas direcciones de correo electrónico contienen la cadena de consulta (email__icontains=query). Los resultados se serializan utilizando un objeto UserSerializer en modo de lista (many=True), y la respuesta se estructura como un diccionario con la clave 'users' que contiene los datos serializados de los usuarios encontrados.

```python
@api_view(['GET'])
def search(request):
    query = request.query_params.get('query')
    if query is None:
        query = ''
    user = User.objects.filter(email__icontains=query)
    serializer = UserSerializer(user, many=True)
    return Response({'users': serializer.data})
```

API 4: Eliminar usuario
Esta API permite eliminar un usuario específico mediante una solicitud DELETE. El parámetro pk en la URL identifica al usuario que se desea eliminar. La API recupera al usuario correspondiente desde la base de datos.
La eliminación del usuario solo está permitida si el usuario que realiza la solicitud tiene el rol de administrador (request.user.is_staff). En caso de que el usuario no tenga los permisos necesarios, la API responde con un código de estado 401 (Unauthorized).
Si el usuario que realiza la solicitud es un administrador, se procede a eliminar el usuario de la base de datos (user.delete()). La API responde con un código de estado 204 (No Content) indicando que la operación fue exitosa y que no hay contenido adicional que se deba enviar como respuesta.

```python
@api_view(['DELETE'])
def delete_user(request, pk):
    user = User.objects.get(pk=pk)
    if request.user.is_staff:
        user.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
    return Response(status=status.HTTP_401_UNAUTHORIZED)
```

API 5: Obtener lista de usuarios
Esta API permite obtener una lista de usuarios mediante una solicitud GET. Solo los usuarios con el rol de administrador (request.user.is_staff) tienen acceso a esta funcionalidad. En caso de que el usuario que realiza la solicitud no sea un administrador, la API responde con un código de estado 401 (Unauthorized).
Si la solicitud proviene de un administrador, la API obtiene la lista de usuarios excluyendo al usuario con la dirección de correo electrónico 'admin@admin.com'. A continuación, se utiliza un serializador (UserSerializer) para convertir los datos de los usuarios en un formato adecuado para su transmisión. La respuesta se compone de los datos serializados de la lista de usuarios y se envía como respuesta.

```
@api_view(['GET'])
def get_users(request):
    if request.user.is_staff:
        users = User.objects.exclude(email='admin@admin.com')
        serializer = UserSerializer(users, many=True)
        return Response(serializer.data)
    return Response(serializer.data, status=status.HTTP_401_UNAUTHORIZED)
```

API 6: Registro de nuevos usuarios
Esta API permite registrar nuevos usuarios mediante una solicitud POST. La información del usuario a registrar se obtiene desde la solicitud (request.data). Se crea un nuevo usuario en la base de datos utilizando los datos proporcionados, incluyendo el correo electrónico, nombre, apellido y una contraseña cifrada utilizando la función make_password para mayor seguridad.
A continuación, se utiliza un serializador especializado para el registro de usuarios (RegisterUserSerializer) para convertir los datos del usuario recién creado en un formato adecuado para ser transmitido como respuesta. El serializador se configura para manejar un solo objeto (many=False), ya que se trata de la creación de un único usuario.

```
@api_view(['POST'])
def register(request):
    data = request.data
    user = User.objects.create(
        email=data['email'],
        name=data['name'],
        last_name=data['last_name'],
        password=make_password(data['password'])
    )
    serializer = RegisterUserSerializer(user, many=False)
    return Response(serializer.data)
```

II.- Aplicación de Productos (products).

A.- Base de datos; Creación de tabla(s) (Models.py).

En esta aplicación, a diferencia de Users, se utilizan dos tablas para almacenar datos. Una de los productos per se y otra para los reviews de dicho producto. Dichas tablas están conectadas por medio de una *Foreign Key*.

Clase '*Products*':

```
class Product(models.Model):
    slug = models.SlugField(max_length=50, null=True, blank=True)
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    name = models.CharField(max_length=100, blank=True)
    image = models.ImageField(default='placeholder.png')
    category = models.CharField(max_length=100, blank=True)
    description = models.CharField(max_length=100, blank=True)
    rating = models.DecimalField(max_digits=10,
                                 decimal_places=2,
                                 null=True, blank=True)
    num_reviews = models.IntegerField(default=0)
    price = models.DecimalField(max_digits=10,
                                decimal_places=2,
                                null=True, blank=True)
    count_in_stock = models.IntegerField(default=0)
    created = models.DateTimeField(auto_now_add=True)
```

Dentro de la creación de la tabla Product, se definen los atributos que caracterizan a cada producto en la base de datos. A continuación, una breve explicación de cada atributo:

- slug: Objeto de tipo texto (string) que almacena una versión amigable del nombre del producto para su uso en URLs. Puede tener un máximo de 50 caracteres y es opcional.
- user: Clave foránea que establece una relación con el modelo de usuario (User). Indica el usuario asociado a este producto. En caso de que el usuario se elimine, se establece como nulo (on_delete=models.SET_NULL).
- name: Objeto de tipo texto (string) que almacena el nombre del producto. Puede tener un máximo de 100 caracteres.
- image: Objeto de imagen que permite almacenar la imagen asociada al producto. En caso de no proporcionar una imagen, se guarda una imagen por defecto ('placeholder.png').
- category: Objeto de tipo texto (string) que almacena la categoría a la que pertenece el producto. Puede tener un máximo de 100 caracteres.
- description: Objeto de tipo texto (string) que almacena la descripción del producto. Puede tener un máximo de 100 caracteres.
- rating: Objeto decimal que almacena la calificación del producto. Permite hasta 10 dígitos en total y 2 decimales. Es opcional y puede ser nulo.
- num_reviews: Objeto entero que almacena la cantidad de reseñas del producto. Se establece un valor predeterminado de 0.

- price: Objeto decimal que almacena el precio del producto. Permite hasta 10 dígitos en total y 2 decimales. Es opcional y puede ser nulo.

- count_in_stock: Objeto entero que almacena la cantidad disponible en stock del producto. Se establece un valor predeterminado de 0.
- created: Objeto de tipo fecha y hora que almacena la fecha de creación del producto. Se actualiza automáticamente con la fecha y hora actual cuando se crea el producto.

Clase '*Reviews*':

```
class Reviews(models.Model):
    product = models.ForeignKey(Product, on_delete=models.SET_NULL, null=True)
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    rating = models.DecimalField(max_digits=10,
                                 decimal_places=2,
                                 null=True, blank=True)
    description = models.CharField(max_length=100, blank=True)
    created = models.DateTimeField(auto_now_add=True)
```

Dentro de la creación de la tabla Reviews, se definen los atributos que caracterizan a cada reseña en la base de datos. A continuación, una breve explicación de cada atributo:

- product: Clave foránea que establece una relación con el modelo Product. Indica el producto asociado a esta reseña. En caso de que el producto se elimine, se establece como nulo (on_delete=models.SET_NULL).
- user: Clave foránea que establece una relación con el modelo User. Indica el usuario asociado a esta reseña. En caso de que el usuario se elimine, se establece como nulo (on_delete=models.SET_NULL).
- rating: Objeto decimal que almacena la calificación de la reseña. Permite hasta 10 dígitos en total y 2 decimales. Es opcional y puede ser nulo.
- description: Objeto de tipo texto (string) que almacena la descripción de la reseña. Puede tener un máximo de 100 caracteres.
- created: Objeto de tipo fecha y hora que almacena la fecha de creación de la reseña. Se actualiza automáticamente con la fecha y hora actual cuando se crea la reseña.

Con esta estructura de base de datos, la API puede trabajar de manera eficiente al manipular y presentar información sobre los productos disponibles.

B.- Conexión Backend/Frontend (Serializers.py):

```
class ReviewSerializer(serializers.ModelSerializer):
    avatar = serializers.SerializerMethodField(source='user.avatar.url')
    user = serializers.ReadOnlyField(source='user.email')

    class Meta:
        model = Reviews
        fields = "__all__"

    def get_avatar(self, obj):
        return obj.user.avatar.url


class ProductSerializer(serializers.ModelSerializer):
    reviews = serializers.SerializerMethodField(read_only=True)

    class Meta:
        model = Product
        fields = "__all__"

    def get_reviews(self, obj):
        reviews = obj.reviews_set.all()
        serializer = ReviewSerializer(reviews, many=True)
        return serializer.data
```

El archivo de serializers contiene dos clases, ReviewSerializer y ProductSerializer, que definen la lógica para serializar los modelos Reviews y Product, respectivamente.
El propósito principal de este archivo es proporcionar lógica de serialización para los modelos Reviews y Product. La serialización es el proceso de convertir objetos complejos, como modelos de base de datos, en formatos que se pueden representar y transmitir de manera eficiente, como JSON. Estos serializers facilitan la presentación de datos en las respuestas de la API.
Estos serializers se utilizan en conjunto con las funciones del archivo Views.py para dar lógica a los endpoints de la API. Permiten la correcta presentación de la información en las respuestas y la interpretación de los datos recibidos en las solicitudes.

API 1: Crear reseña de producto

Esta API permite a usuarios autenticados crear reseñas para productos específicos mediante una solicitud POST. Se requiere que el usuario esté autenticado, lo cual es gestionado por el decorador @permission_classes([IsAuthenticated]). La información de la reseña se recibe en el cuerpo de la solicitud y se valida utilizando un serializador especializado llamado ReviewSerializer.

El identificador único del producto al que se asociará la reseña se obtiene a través del parámetro de ruta pk. Se realiza una consulta a la base de datos para recuperar el objeto de producto correspondiente. Una vez validada la información de la reseña, se guarda en la base de datos, asociándola al usuario autenticado y al producto específico.

Si la validación es exitosa, la API responde con los datos de la reseña recién creada en formato serializado y devuelve un código de estado HTTP 201 (CREATED). En caso de que la validación falle, la API responde con los datos de error y un código de estado HTTP 400 (BAD REQUEST).

```python
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def create_review(request, pk):
    serializer = ReviewSerializer(data=request.data)
    product = Product.objects.get(pk=pk)
    if serializer.is_valid():
        serializer.save(user=request.user, product=product)
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    else:
        return Response(serializer.data, status=status.HTTP_400_BAD_REQUEST)
```

API 2: Obtener productos por categoría

Esta API permite a los usuarios obtener una lista de productos filtrados por categoría mediante una solicitud GET. El parámetro de ruta category especifica la categoría por la cual se deben filtrar los productos. Al recibir la solicitud, se realiza una consulta a la base de datos para recuperar todos los productos que pertenecen a la categoría proporcionada.

Los datos de los productos obtenidos se serializan utilizando un serializador específico llamado ProductSerializer, configurado para manejar múltiples objetos (many=True), ya que se espera una lista de productos en lugar de un solo objeto. La API responde con la lista de productos filtrados en formato serializado.

```python
@api_view(['GET'])
def get_prod_by_cate(request, category):
    products = Product.objects.filter(category=category)
    serializer = ProductSerializer(products, many=True)
    return Response(serializer.data)
```

API 3: Búsqueda de productos

Esta API permite a los usuarios realizar búsquedas de productos mediante una solicitud GET. La búsqueda se realiza utilizando el parámetro de consulta query, el cual contiene el término de búsqueda proporcionado por el usuario. En caso de que no se proporcione un término de búsqueda, se establece una cadena vacía como valor predeterminado para evitar errores.

La API realiza una consulta a la base de datos utilizando el método filter en el modelo de Producto, buscando aquellos cuyos nombres contengan la cadena de búsqueda (case-insensitive) utilizando el campo name__icontains. Los productos resultantes se serializan con el ProductSerializer, configurado para manejar múltiples objetos.

La respuesta de la API incluye un diccionario con la clave 'products', que contiene la lista de productos filtrados por la cadena de búsqueda en formato serializado.

```python
@api_view(['GET'])
def search(request):
    query = request.query_params.get('query')
    if query is None:
        query = ''
    product = Product.objects.filter(name__icontains=query)
    serializer = ProductSerializer(product, many=True)
    return Response({'products': serializer.data})
```

API 4: Obtener lista paginada de productos

Esta API proporciona una lista paginada de todos los productos disponibles mediante una solicitud GET. Se realiza una consulta a la base de datos para recuperar todos los productos almacenados en el modelo de Producto. La paginación se gestiona mediante una instancia de la clase CustomPagination, que es responsable de dividir la lista completa de productos en páginas basadas en la configuración de paginación y la solicitud del cliente.

La lista paginada de productos se obtiene utilizando el método paginate_queryset de la clase de paginación, que toma la lista completa de productos y la información de la solicitud para determinar qué segmento de la lista debe incluirse en la respuesta.

Los productos paginados se serializan utilizando el ProductSerializer configurado para manejar múltiples objetos. La respuesta de la API incluye la lista de productos paginados en formato serializado, proporcionando detalles como el nombre del producto, descripción, precio, etc.

```python
@api_view(['GET'])
def get_products(request):
    products = Product.objects.all()
    paginator = CustomPagination()
    paginated_products = paginator.paginate_queryset(products, request)
    serializer = ProductSerializer(paginated_products, many=True)
    return paginator.get_paginated_response(serializer.data)
```

API 5: Obtener detalles de producto para administradores
Esta API permite a los administradores obtener detalles específicos de un producto mediante una solicitud GET. El identificador único del producto se obtiene a través del parámetro de ruta id. Se realiza una consulta a la base de datos para recuperar el objeto de producto correspondiente utilizando el método get en lugar de filter ya que se espera un único objeto.
Los detalles del producto se serializan utilizando el ProductSerializer configurado para manejar un solo objeto (many=False), ya que se trata de la obtención de información detallada de un único producto.
La API responde con los detalles del producto solicitado en formato serializado.

```
@api_view(['GET'])
def get_product_admin(request, id):
    products = Product.objects.get(id=id)
    serializer = ProductSerializer(products, many=False)
    return Response(serializer.data)
```

API 6: Obtener detalles de producto por slug
Esta API permite a los usuarios obtener detalles específicos de un producto mediante una solicitud GET. El identificador único del producto se obtiene a través del parámetro de ruta slug, que generalmente es una versión amigable y legible por humanos del nombre del producto. Se realiza una consulta a la base de datos para recuperar el objeto de producto correspondiente utilizando el campo slug.
Los detalles del producto se serializan utilizando el ProductSerializer configurado para manejar un solo objeto (many=False), ya que se trata de la obtención de información detallada de un único producto.
La API responde con los detalles del producto solicitado en formato serializado. Utilizar el slug como identificador proporciona una URL más amigable y fácil de recordar para los usuarios al acceder a información detallada de productos en la aplicación.

```
@api_view(['GET'])
def get_product(request, slug):
    products = Product.objects.get(slug=slug)
    serializer = ProductSerializer(products, many=False)
    return Response(serializer.data)
```

API 7: Crear nuevo producto (para administradores)
Esta API permite a los administradores crear nuevos productos mediante una solicitud POST. Antes de proceder con la creación, se verifica que el usuario que realiza la solicitud sea un administrador, comprobando la propiedad is_staff del usuario autenticado.
Se utiliza un serializador específico llamado ProductSerializer para validar y procesar la información del nuevo producto a partir de la solicitud (request.data). Si la validación es exitosa, se genera un campo slug combinando el nombre y la categoría del producto y se utiliza la función slugify para convertirlo en una versión amigable para URL.
Luego, se verifica si ya existe un producto con el mismo slug en la base de datos. Si existe, se responde con un código de estado HTTP 400 (BAD REQUEST). En caso contrario, se guarda el nuevo producto en la base de datos, asociándolo al usuario administrador que lo creó.
Si la solicitud es realizada por un usuario no autorizado (que no es administrador), se responde con un código de estado HTTP 401 (UNAUTHORIZED). La API responde con los detalles del producto recién creado en formato serializado y el código de estado HTTP 201 (CREATED) en caso de éxito, o con detalles de error y el código de estado HTTP 400 (BAD REQUEST) en caso de fallo en la validación del serializador.

```
@api_view(['POST'])
def create_product(request):
    if request.user.is_staff:
        serializer = ProductSerializer(data=request.data)
        if serializer.is_valid():
            name = serializer.validated_data['name']
            category = serializer.validated_data['category']
            s = name + category
            slug = slugify(s)
            if serializer.Meta.model.objects.filter(slug=slug).exists():
                return Response(serializer.data, status=status.HTTP_400_BAD_REQUEST)
            serializer.save(user=request.user, slug=slug)
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.data, status=status.HTTP_400_BAD_REQUEST)
    else:
        return Response(serializer.data, status=status.HTTP_401_UNAUTHORIZED)
```

API 8: Editar información de producto (para administradores)
Esta API permite a los administradores editar la información de un producto existente mediante una solicitud PUT. Se utiliza el identificador único del producto, pk, para recuperar el objeto de producto correspondiente de la base de datos.
Antes de realizar la edición, se verifica que el usuario que realiza la solicitud sea un administrador, comprobando la propiedad is_staff del usuario autenticado.
Se utiliza un serializador específico llamado ProductSerializer para validar y procesar la información actualizada del producto a partir de la solicitud (request.data). Si la validación es exitosa, se genera un nuevo campo slug combinando el nombre y la categoría del producto y se utiliza la función slugify para convertirlo en una versión amigable para URL.
Se guarda la información actualizada en la base de datos, asociando al usuario administrador que realizó la edición. La API responde con los detalles del producto editado en formato serializado.
Si la solicitud es realizada por un usuario no autorizado (que no es administrador), la API responde con un código de estado HTTP 401 (UNAUTHORIZED). En caso de fallo en la validación del serializador, se responde con un código de estado HTTP 400 (BAD REQUEST).

```
@api_view(['PUT'])
def edit_product(request, pk):
    product = Product.objects.get(pk=pk)
    if request.user.is_staff:
        serializer = ProductSerializer(product, data=request.data)
        if serializer.is_valid():
            name = serializer.validated_data['name']
            category = serializer.validated_data['category']
            s = name + category
            slug = slugify(s)
            serializer.save(user=request.user, slug=slug)
            return Response(serializer.data)
        return Response(status=status.HTTP_400_BAD_REQUEST)
    else:
        return Response(status=status.HTTP_401_UNAUTHORIZED)
```

API 9: Eliminar producto (para administradores)
Esta API permite a los administradores eliminar un producto existente mediante una solicitud DELETE. Se utiliza el identificador único del producto, pk, para recuperar el objeto de producto correspondiente de la base de datos.
Antes de realizar la eliminación, se verifica que el usuario que realiza la solicitud sea un administrador, comprobando la propiedad is_staff del usuario autenticado.
Si la verificación es exitosa, se procede a eliminar el producto de la base de datos utilizando el método delete(). La API responde con un código de estado HTTP 204 (NO CONTENT) para indicar que la eliminación se realizó con éxito y no hay contenido adicional en la respuesta.
Si la solicitud es realizada por un usuario no autorizado (que no es administrador), la API responde con un código de estado HTTP 401 (UNAUTHORIZED) para indicar que el usuario no tiene los permisos necesarios para realizar la acción de eliminación.

```python
@api_view(['DELETE'])
def delete_product(request, pk):
    product = Product.objects.get(pk=pk)
    if request.user.is_staff:
        product.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
    else:
        return Response(status=status.HTTP_401_UNAUTHORIZED)
```

III.- Aplicación de Ordenes (orders).

A.- Base de datos; Creación de tabla(s) (models.py)

```python
class Order(models.Model):
    user = models.ForeignKey(User, on_delete=models.SET_NULL, null=True)
    total_price = models.CharField(max_length=250, blank=True)
    is_delivered = models.BooleanField(default=False)
    delivered_at = models.DateTimeField(auto_now_add=False, null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)


class Orderitem(models.Model):
    product = models.ForeignKey(Product, on_delete=models.SET_NULL, null=True)
    order = models.ForeignKey(Order, on_delete=models.SET_NULL, null=True)
    quantity = models.IntegerField(null=True, blank=True, default=0)
    price = models.CharField(max_length=250, blank=True)


class ShippingAddress(models.Model):
    order = models.OneToOneField(Order, on_delete=models.CASCADE, null=True, blank=True)
    address = models.CharField(max_length=250, blank=True)
    city = models.CharField(max_length=100, blank=True)
    postal_code = models.CharField(max_length=100, blank=True)
```

Dentro de la creación de las tablas Order, Orderitem y ShippingAddress en el archivo models.py de la aplicación 'orders', se definen los atributos que caracterizan a cada entidad en la base de datos. A continuación, una breve explicación de cada modelo:
Clase Order:

- user: Clave foránea que establece una relación con el modelo User. Indica el usuario asociado a esta orden. En caso de que el usuario se elimine, se establece como nulo (on_delete=models.SET_NULL).
- total_price: Campo de texto que almacena el precio total de la orden con un máximo de 250 caracteres. Es opcional y puede estar en blanco.
- is_delivered: Campo booleano que indica si la orden ha sido entregada o no. Se establece como falso por defecto.
- delivered_at: Campo de fecha y hora que almacena la fecha de entrega de la orden. Es opcional y puede ser nulo.
- created_at: Campo de fecha y hora que almacena la fecha de creación de la orden. Se actualiza automáticamente con la fecha y hora actual cuando se crea la orden.

Clase Orderitem:

- product: Clave foránea que establece una relación con el modelo Product. Indica el producto asociado a este ítem de orden. En caso de que el producto se elimine, se establece como nulo (on_delete=models.SET_NULL).
- order: Clave foránea que establece una relación con el modelo Order. Indica la orden a la que pertenece este ítem. En caso de que la orden se elimine, se establece como nulo (on_delete=models.SET_NULL).
- quantity: Campo entero que almacena la cantidad de productos en este ítem de orden. Es opcional y puede ser nulo, con un valor predeterminado de cero.
- price: Campo de texto que almacena el precio del ítem de orden con un máximo de 250 caracteres. Es opcional y puede estar en blanco.

Clase ShippingAddress:

- order: Clave foránea que establece una relación uno a uno con el modelo Order. Indica la orden asociada a esta dirección de envío. En caso de que la orden se elimine, también se elimina esta dirección (on_delete=models.CASCADE).
- address: Campo de texto que almacena la dirección de envío con un máximo de 250 caracteres. Es opcional y puede estar en blanco.
- city: Campo de texto que almacena la ciudad de envío con un máximo de 100 caracteres. Es opcional y puede estar en blanco.
- postal_code: Campo de texto que almacena el código postal de envío con un máximo de 100 caracteres. Es opcional y puede estar en blanco.

Con esta estructura de base de datos, la API puede manejar eficientemente la información relacionada con las órdenes, ítems de orden y direcciones de envío asociadas.

B.- Conexión Backend/Frontend (Serializers.py):

```python
class ShippingSerializer(serializers.ModelSerializer):
    class Meta:
        model = ShippingAddress
        fields = '__all__'


class OrderItemSerializer(serializers.ModelSerializer):

    product = serializers.ReadOnlyField(source='product.name')

    class Meta:
        model = Orderitem
        fields = '__all__'


class OrderSerializer(serializers.ModelSerializer):

    user = serializers.ReadOnlyField(source='user.email')
    order_items = serializers.SerializerMethodField(read_only=True)
    shipping_address = serializers.SerializerMethodField(read_only=True)

    class Meta:
        model = Order
        fields = '__all__'

    def get_order_items(self, obj):
        items = obj.orderitem_set.all()
        serializer = OrderItemSerializer(items, many=True)
        return serializer.data

    def get_shipping_address(self, obj):
        try:
            address = ShippingSerializer(
                obj.shippingaddress, many=False).data
        except:
            address = False
        return address
```

El archivo serializers.py de la aplicación Orders presenta tres clases esenciales: ShippingSerializer, OrderItemSerializer y OrderSerializer, cada una encargada de definir la lógica de serialización para los modelos correspondientes: ShippingAddress, Orderitem y Order, respectivamente.

En primer lugar, el ShippingSerializer se encarga de la serialización del modelo ShippingAddress, definiendo los campos a incluir en el proceso de transformación de objetos complejos a un formato representable y eficientemente transmisible.

A continuación, el OrderItemSerializer desempeña un papel clave en la serialización del modelo Orderitem. Además de los campos generales definidos en el modelo, introduce la lógica necesaria para presentar de manera legible la información del producto asociado a cada ítem de orden, utilizando un campo de solo lectura.

Por último, el OrderSerializer aborda la serialización del modelo Order. Este serializer incluye campos como el usuario, los elementos de la orden y la dirección de envío. Se complementa con funciones personalizadas como get_order_items y get_shipping_address, las cuales utilizan otros serializers definidos anteriormente para asegurar la presentación coherente de la información en las respuestas de la API.

Estos serializers se integran armónicamente con las funciones del archivo Views.py para dotar de lógica a los puntos finales de la API de Orders. No solo facilitan la correcta presentación de la información en las respuestas generadas, sino que también permiten la interpretación precisa de los datos recibidos en las solicitudes, contribuyendo así a la eficiencia y coherencia del sistema.

C.- Requests/APIs (Views,py):

API 1: Búsqueda de pedidos (para administradores)
Esta API permite a los administradores buscar pedidos utilizando una solicitud GET. El permiso necesario para acceder a esta API está configurado mediante el decorador @permission_classes([IsAdminUser]), lo que garantiza que solo los usuarios administradores puedan realizar la búsqueda.

La búsqueda se realiza utilizando el parámetro de consulta query, que contiene el término de búsqueda proporcionado por el usuario. Si no se proporciona un término de búsqueda, se establece una cadena vacía como valor predeterminado.
La API realiza una consulta a la base de datos utilizando el modelo de Pedido (Order), filtrando los pedidos cuyo usuario tenga una dirección de correo electrónico que contenga la cadena de búsqueda (case-insensitive).

Los resultados se serializan utilizando el OrderSerializer configurado para manejar múltiples objetos. La respuesta de la API incluye un diccionario con la clave 'orders', que contiene la lista de pedidos filtrados por la cadena de búsqueda en formato serializado.

```python
@api_view(['GET'])
@permission_classes([IsAdminUser])
def search(request):
    query = request.query_params.get('query')
    if query is None:
        query = ''
    order = Order.objects.filter(user__email__icontains=query)
    serializer = OrderSerializer(order, many=True)
    return Response({'orders': serializer.data})
```

API 2: Obtener lista de pedidos (para administradores)
Esta API permite a los administradores obtener una lista de todos los pedidos mediante una solicitud GET. El permiso necesario para acceder a esta API está configurado mediante el decorador @permission_classes([IsAdminUser]), asegurando que solo los usuarios administradores puedan realizar la solicitud.
Se realiza una consulta a la base de datos para recuperar todos los pedidos almacenados en el modelo de Pedido (Order). Los resultados se serializan utilizando el OrderSerializer configurado para manejar múltiples objetos (many=True), ya que se espera una lista de pedidos en lugar de un solo objeto.

```python
@api_view(['GET'])
@permission_classes([IsAdminUser])
def get_orders(request):
    orders = Order.objects.all()
    serializer = OrderSerializer(orders, many=True)
    return Response(serializer.data)
```

API 3: Crear nuevo pedido
Esta API permite a usuarios autenticados crear un nuevo pedido mediante una solicitud POST. El permiso necesario para acceder a esta API está configurado mediante el decorador @permission_classes([IsAuthenticated]), asegurando que solo los usuarios autenticados puedan realizar la solicitud.
La información necesaria para crear el pedido se obtiene de la solicitud (request.data), incluyendo los elementos del pedido (order_items), el precio total (total_price), la dirección de envío (address), la ciudad (city), y el código postal (postal_code).
Se verifica que el precio total proporcionado coincida con la suma de los precios de los elementos del pedido. En caso afirmativo, se procede a crear el pedido y asociar los detalles de envío y los elementos del pedido.
Para cada elemento del pedido, se reduce la cantidad en stock del producto correspondiente en la base de datos. La API responde con los detalles del pedido recién creado en formato serializado y un código de estado HTTP 201 (CREATED) en caso de éxito.
En caso de que el precio total no coincida con la suma de los precios de los elementos del pedido, la API responde con un mensaje de error y un código de estado HTTP 400 (BAD REQUEST). Esta API es esencial para permitir a los usuarios realizar nuevos pedidos en la aplicación.

```python
@api_view(['POST'])
@permission_classes([IsAuthenticated])
def create_order(request):
    user = request.user
    data = request.data
    orderItems = data['order_items']
    total_price = data['total_price']

    sum_of_prices = sum(int(float(item['price'])) * item['quantity'] for item in orderItems)

    if total_price == sum_of_prices:
        order = Order.objects.create(
            user=user,
            total_price=total_price
        )

        ShippingAddress.objects.create(
            order=order,
            address=data['address'],
            city=data['city'],
            postal_code=data['postal_code'],
        )

        for i in orderItems:
            product = Product.objects.get(id=i['id'])
            item = OrderItem.objects.create(
                product=product,
                order=order,
                quantity=i['quantity'],
                price=i['price']
            )

            product.count_in_stock -= item.quantity
            product.save()

        serializer = OrderSerializer(order, many=False)
        return Response(serializer.data, status=status.HTTP_201_CREATED)
    else:
```

API 4: Obtener detalles de un pedido específico
Esta API permite a usuarios autenticados obtener detalles específicos de un pedido mediante una solicitud GET. El permiso necesario para acceder a esta API está configurado mediante el decorador @permission_classes([IsAuthenticated]), asegurando que solo los usuarios autenticados puedan realizar la solicitud.
Se utiliza el identificador único del pedido, pk, para recuperar el objeto de pedido correspondiente de la base de datos. Se verifica si el usuario autenticado es un administrador o el usuario asociado al pedido.
Si el usuario tiene los permisos necesarios, se serializan los detalles del pedido utilizando el OrderSerializer configurado para manejar un solo objeto (many=False). La API responde con los detalles del pedido en formato serializado.
En caso de que el usuario no tenga acceso para ver el pedido, la API responde con un mensaje de error y un código de estado HTTP 401 (UNAUTHORIZED). Si el pedido no existe, la API responde con un mensaje de error y un código de estado HTTP 400 (BAD REQUEST).

```python
@api_view(['GET'])
@permission_classes([IsAuthenticated])
def solo_order(request, pk):
    user = request.user
    try:
        order = Order.objects.get(pk=pk)
        if user.is_staff or order.user == user:
            serializer = OrderSerializer(order, many=False)
            return Response(serializer.data)
        else:
            Response({'detail': 'No access to view orders'},
                     status=status.HTTP_401_UNAUTHORIZED)
    except:
        return Response({'detail': 'Order does not exist'}, status=status.HTTP_400_BAD_REQUEST)
```

API 5: Obtener mis pedidos

Esta API permite a usuarios autenticados obtener una lista de sus propios pedidos mediante una solicitud GET. El permiso necesario para acceder a esta API está configurado mediante el decorador @permission_classes([IsAuthenticated]), asegurando que solo los usuarios autenticados puedan realizar la solicitud.

Se obtiene el usuario autenticado a partir de la solicitud (request.user) y se realiza una consulta a la base de datos para recuperar todos los pedidos asociados a ese usuario utilizando la relación inversa order_set.

Los resultados se serializan utilizando el OrderSerializer configurado para manejar múltiples objetos (many=True), ya que se espera una lista de pedidos en lugar de un solo objeto. La API responde con la lista de pedidos del usuario en formato serializado.

```python
@api_view(['GET'])
@permission_classes([IsAuthenticated])
def my_orders(request):
    user = request.user
    orders = user.order_set.all()
    serializer = OrderSerializer(orders, many=True)
    return Response(serializer.data)
```

API 6: Marcar pedido como entregado (para administradores)

Esta API permite a los administradores marcar un pedido como entregado mediante una solicitud PUT. El permiso necesario para acceder a esta API está configurado mediante el decorador @permission_classes([IsAdminUser]), asegurando que solo los usuarios administradores puedan realizar la acción.

Se utiliza el identificador único del pedido, pk, para recuperar el objeto de pedido correspondiente de la base de datos. Luego, se actualiza el estado del pedido estableciendo la propiedad is_delivered a True y se registra la fecha y hora de la entrega en delivered_at. Finalmente, se guarda la actualización en la base de datos.

```python
@api_view(['PUT'])
@permission_classes([IsAdminUser])
def delivered(request, pk):
    order = Order.objects.get(pk=pk)
    order.is_delivered = True
    order.delivered_at = datetime.now()
    order.save()
    return Response('Order was delivered')
```

IV.- Conclusión:

En base a este documento se presentó una descripción entendible del funcionamiento de cada API habida en nuestro proyecto. Partiendo de la creación de las tablas en el archivo de modelos y su funcionamiento por medio de serializadores.

Todo esto con la finalidad de tener un entendimiento preciso del funcionamiento de estas API y ayudar a que se facilite la edición o lógica detrás si llega a hacer falta un mantenimiento a dicha lógica.