

Autenticación por Token

Importante

Un token viene a ser como un cifrado de una firma digital pero ¿qué es una firma digital?
Pues vendría siendo algo así como una cadena larga de texto donde está encriptada cierta información

Dentro de un token vamos a guardar o recibir información de un usuario, un token nunca se almacena en un servidor, por lo que no ocupa espacios con sesiones activas

Usualmente en la autenticación normal se consume memoria en el servidor pero si existen muchos usuarios a la vez ingresando pues se cae el servidor, se podría pensar en incrementar la memoria del mismo pero esto es costoso y difícilmente escalable.

La autenticación por token surge como una solución puesto que al logearse lo que se genera es un token, y una vez obtenido el mismo pues es enviado al servidor donde ahí es descifrado y una vez descifrado sabemos a qué usuario pertenece el mismo.

Pero, ¿qué sucede con el usuario?

Pues al momento que se requiere hacer consultas, listar tareas, personas, etc pues la URL va a tener el token y el servidor recibirá la URL con el token y únicamente para la consulta que sea requerida va a transformar el token en los datos del usuario.

Entonces, como para esa consulta ya tenemos el usuario en base al token que nos enviaron, es bastante sencillo tomar ese valor y luego hacer el filtro

Lo que quiere decir que ahora para cada consulta que necesitemos verificación de usuario, necesitamos obligatoriamente enviar un token, pero esto es trabajo en sí del cliente/navegador



Comenzaríamos desde el cliente, haciendo una petición POST para enviar el usuario y contraseña, y realizar el proceso de login.

Se comprobaría que ese usuario y su contraseña son correctos, y de serlos, generar el token JWT para devolverlo al usuario.

A partir de ahí la aplicación cliente, con ese token, haría peticiones solicitando recursos, siempre con ese token JWT dentro de un encabezado, que sería Authorization: Bearer XXXXXXXX, siendo Bearer el tipo de prefijo seguido de todo el contenido del token.

En el servidor se comprobaría el token mediante la firma, para verificar que el token es seguro, y, por tanto podemos confiar en el usuario.

Dentro del cuerpo del token, además, tenemos los datos de quién es el usuario que ha realizado esa petición, porque podemos contener en el payload todos los datos de usuario que queramos.

Tras verificar que el token es correcto y saber quién es el que ha hecho la petición, podemos aplicar entonces el mecanismo de control de acceso, saber si puede acceder o no, y si es así, responder con el recurso protegido, de manera que lo podría recibir de una forma correcta.

De esta forma podríamos implementar el proceso de autenticación, y hacerlo, además, con estos JSON Web Token.

La autenticación basada en tokens es el proceso de verificar la identidad mediante la comprobación de un token. En la gestión de acceso, los servidores utilizan la autenticación por token para comprobar la identidad de un usuario, una API, un ordenador u otro servidor. Un token es un elemento simbólico que expide una fuente de confianza. Pensemos en cómo los policías llevan consigo una insignia expedida por las autoridades que legitima su autoridad. Las fichas pueden ser físicas (como una llave USB) o digitales (un mensaje generado por ordenador o una firma digital).

¿Cómo funciona la autenticación mediante token web?

Un token web es digital, no un objeto físico. Es un mensaje enviado desde un servidor a un cliente y que este almacena temporalmente. El cliente incluye una copia del token en las siguientes solicitudes enviadas al servidor para confirmar el estado de autenticación del cliente.

Mientras que la autenticación con tokens físicos verifica la identidad durante el proceso de inicio de sesión, los tokens web se emiten como resultado de un inicio de sesión con éxito. Mantienen activa la sesión iniciada.

Sin embargo, utilizar tokens web para las sesiones de usuario no siempre es lo mejor. Muchos desarrolladores son partidarios de utilizar cookies en su lugar. Los tokens web se pueden utilizar mejor para la autenticación de puntos finales de la API o para validar una conexión entre servidores, en lugar de entre el servidor y el cliente.

Ojo que nos estamos apoyando en Google Firebase para poder generar los Token

Creación de Proyecto Firebase - Authentication

En firebase se creó el proyecto con el nombre Django-pro , se configuró que la autenticación sea por medio del correo, se creo un template basico con un botón para el login....

Lo que se va a hacer ahora es utilizar la documentación que nos ofrece Firebase e implementar la funcionalidad en este botón, de tal forma que cuando alguien le dé clic en el mismo, este nos abre el formulario para poner nuestros datos de Google y que este a su vez recupere un token.

Ese token es el que debemos enviar hacia nuestro servidor y hacer el proceso respectivo

Integrando Firebase en un template Django

Los servicios de Firebase (como Cloud Firestore, Authentication, Realtime Database, Remote Config y muchos más) están disponibles para importarse en subpaquetes individuales.

<https://www.youtube.com/watch?v=tn5n8zOHHN8> <= Actualización

Nota: Se recomienda usar la versión 9 del SDK, en especial para las apps de producción. Si necesitas compatibilidad con otras opciones de administración de SDK, como `window.firebase`, consulta [Actualiza de la versión 8 al SDK web modular o Formas alternativas de agregar Firebase](#).

En éste caso sería la versión 8 puesto que la 9 está mas orientada a trabajar con JS modular, Vue, React, entonces vamos a actualizar de la versión 8 al SDK web modular.

Entonces se añaden los scripts en el html junto con la configuracion/keys del proyecto, con ésto ya estaría solucionado el primer error debido a las actualizaciones. ya está instalado correctamente el SDK de firebase.

Sumado a éso hay otro problema de actualización con la función de login, entonces para ello creamos una función y dentro de ésta función irá todo lo relacionado al Manejo del flujo de acceso con el SDK de Firebase, código que ya nos facilita firebase

https://firebase.google.com/docs/auth/web/google-signin?authuser=2#web-version-8_1

```
firebase.auth()
  .signInWithPopup(provider)
  .then((result) => {
    /** @type {firebase.auth.OAuthCredential} */
    var credential = result.credential;

    // This gives you a Google Access Token. You can use it to access the Google API.
    var token = credential.accessToken;
    // The signed-in user info.
    var user = result.user;
    // ...
  }).catch((error) => {
    // Handle Errors here.
    var errorCode = error.code;
    var errorMessage = error.message;
    // The email of the user's account used.
    var email = error.email;
    // The firebase.auth.AuthCredential type that was used.
    var credential = error.credential;
    // ...
  });
```

Una vez hecho ésto ya nos cargará el login de Google, pero es necesario mencionar que firebase únicamente trabaja con localhost en lugar de 127.0.0.1 , es decir así <http://localhost:8000/login/>

Actualización Login con Google <== Ésta clase ya se la realizó pero en youtube

Token de Firebase en un template Django

Para que funcione correctamente se cambió el `signInWithPopup` por el `signInWithRedirect` debido a que arrojaba error

APIView Google Login View

Una vez obtenido un token necesitamos crear un servicio que reciba el mismo, que como se vió en la parte teórica lo descifre para identificar al usuario según lo que nos está enviando Google y en caso de que no exista registrarlo y en caso de que no exista recuperarlo.

Para ello se crea un serializador donde solamente se reciba un token y también se crea una vista a la cual voy a llamar normalmente a mi serializador, sin embargo que tipo de vista uso?

Pues para éste caso usaremos la APIView, vista de DRF

Ahora dentro de ésta vista nos tocaría escribir una función de tal manera que se procesa toda la información para el propósito que queremos que es descenciptar el token para saber qué usuario está encriptado

Método POST dentro del ApiView <https://www.cdrf.co/>

En ésta clase se implementará el proceso para descenciptar el token

De igual manera que los otros view, este ApiView viene con algunos métodos que podemos sobrescribir, en éste caso sobrescribiremos el método POST

Entonces primero recupero la info que nos mandan a través del serializador creando una variable serializer

Entonces con éste serializer_class serializamos la data que nos están mandando por el método, en éste caso la data es igual a (request => info enviada por protocolo HTTP)

Una vez que ya tenemos serializados todos los datos que nos han enviado pues toca verificar que la info sea correcta, entonces SOLAMENTE SI ES CORRECTA PODEMOS CONTINUAR HACIENDO PROCESOS

Pregunto si el serializador es valid con el is_valid Ojo que si no es valido el is_Valid nos acepta una excepción

Ahora sí recupero la información => `id_token = serializer.data.get('token_id')`
Y una vez recuperada tengo que descenciptarla

Pero para descenciptarla necesito una Key la cual me facilita el servicio con el que hemos estado trabajando que en éste caso sería Firebase
Lo haremos en la siguiente clase

SDK de Firebase en Python

<https://firebase.google.com/docs/admin/setup?authuser=0&%3Bhl=es&hl=es#python>

Viendo la documentación haremos la respectiva instalación

```
pip install firebase-admin
```

Debemos inicializar Firebase dentro de nuestro servidor Python utilizando las credenciales

que aparentemente debemos descargarla desde nuestro proyecto y nos va a traer un archivo JSON

Una vez obtenidas las credenciales inicializamos la aplicación dentro del proyecto python,

lo cual lo haremos en nuestro entorno local por obvias razones

Función y Método para Decodificar un Token de Firebase

Importamos las librerías de FB en nuestro view y por medio del auth accedo a una función de firebase para decodificar

```
decoded_token = auth.verify_id_token(id_token)
```

Entonces ya con el token decodificado pues ya puedo ir accediendo a los valores de mi usuario, además ya puedo utilizar Django mismo para crear mi objeto usuario en éste caso.

Sin embargo, primero debo comprobar que el usuario ya exista en la BD caso contrario crearlo, para ello nos apoyaremos del método antes visto (Sección modelos avanzados)

get_or_create

```
usuario, created = User.objects.get_or_create(
    email=email, #Especifico en base a que quiero que recupere el usuario
    defaults={ #Aquí añado los valores que no estan en mi vista para que si no encuentra pues lo cree
        'full_name' : name,
        'email' : email,
        'is_active' : True,
        #Los demás atributos no se están poniendo debido a que no son obligatorios
    }
)
```

Pero, ¿cómo implemento esto en un proyecto que no esté usando Django REST Framework?

Si ya tengo el usuario creado simplemente llamo a la función login y le mando como parametro el usuario
login(usuario)

Continuando pues, una vez obtenido el token decodificado y los datos del usuario lo que vamos a hacer es crear un token INTERNO dentro del proyecto de Django.

De tal forma que cada vez que alguien haga login con su cuenta de Google, después de que verifique los datos, recupere se token que lo hemos creado nosotros.

Si en algún momento ya no queremos trabajar con Google, igual seguimos teniendo el token o seguimos nosotros administrando nuestro propio token.

Creación de Token Con Django Rest Framework

Nos apoyaremos de un complemento de DRF, se lo debe añadir en el base.py, éste paquete tiene una particularidad y es que trae sus propios modelos internos por lo que una vez añadido se debe hacer el migrate.

Obviamente si en mi método POST sí recupera el usuario pues ya no necesitaríamos crear un token porque ya existe el usuario y ya está registrado... El token es para usuarios nuevos

Entonces ya sea que el usuario se cree o ya haya existido pues recupero el token pero para qué lo recupero?

Debido a que ahora lo que voy a devolver a mi frontend es el token que hemos creado y administramos nosotros

En pocas palabras, hemos usado el token que nos ha enviado Google pero para DESCIFRAR información, ahora que ya tenemos la información ya podemos hacernos cargo de este usuario por completo

Ahora prosiguiendo lo que se va a hacer es crear un User (se pone userGet haciendo referencia a que es el usuario que se va a devolver) y se manda el diccionario/objeto.

Posteriormente hacemos un return de otro diccionario donde esté el usuario que recuperamos y el Token que acabamos de recuperar o crear.

Para ello usaremos el response de DRF en el cual retornaremos un token el cual hemos creado y el user que también hemos creado

Según lo investigado existe un error en firebase al momento de usar el SignInWithPopUp por lo que se usa el SignInRedirect

Sumado a eso también se recupera el token del id del cliente
<https://firebase.google.com/docs/auth/admin/verify-id-tokens?authuser=0&hl=es#web>

Envío de Token desde un Template Django

Nos apoyaremos de una librería Axios, una vez importada la misma .

Se usará Axios para que envíe el Token tal cual como si lo estuviera haciendo un frontend

Usamos axios.post('mando la url a la que quiero que haga post', 'también mando un serializador con el cual va a ir el post

Sin embargo el serializador debe ir con formato clave-valor entonces se crea una variable donde pongo la clave y el valor y luego a ésta variable le llamo en mi metodo

Token y Json Web Token Teoría

El token generado por Google y el que hemos hecho son diferentes debido a que con el tiempo se mejoró la propuesta de los tokens debido a que si se tiene demasiadas consultas se tendría que hacer peticiones a la BD para que recoja el token y a partir del mismo identifique al usuario.

Entonces para mejorar eso se usa los JSON WEB TOKEN (JWT) que tienen como ventaja que toda la info está dentro del token en sí, no se necesita hacer la petición a la BD solo decodificar el token, por ende es más seguro

Copia de Seguridad con Django

<https://neunapp.com/contenido/crear-copia-de-seguridad-base-de-datos-desde-django-19772>

Crear Copia de Seguridad con Django => `python manage.py dumpdata`

Volvar Copia de Seguridad con Django => `python manage.py loaddata nombre_copia_bd.json`

ApiListView - Lista de Productos por usuario

Se plantea como requerimiento que en el caso que el frontend que está trabajando con nuestro backend nos ha pedido que listemos todos los productos de esta base de datos, lo cual ya sabemos que es un `ListAPIView`, pero sumado a eso con una característica que se filtre por el usuario que el frontend nos está indicando (nos lo indica enviándonos el token en las cabeceras de la consulta)

Entonces primero hago el serializador normalmente, le paso los fields respectivos, luego creo mi view (`ListAPIView`) donde llamo a mi serializador y en el queryset en primera instancia voy a listar todos los productos, de igual forma se añaden las urls respectivamente. En la siguiente clase se añadirá el filtro del requerimiento planteado

Token Authenticate y Permission Class

Si Django no encuentra un user pues lo que hará es mandar un `AnonymousUser`, primero se creará el manager de mi producto que me va a recuperar un usuario recibido como parámetro... sin embargo si no se está iniciada sesión obtengo un `AnonymousUser` y al no tener ID me arroja error.

Entonces para solucionar esto dentro de Django pues recordar que estamos usando `rest_framework.authtoken` y en la documentación nos dice que debemos de hacer uso de `authentication_classes` para poder descencriptar el token y recuperar al usuario

Lo definimos en la vista habiéndolo previamente importado, sumado a esto también añadimos otro atributo.. el `permission_classes` pero con la finalidad de que en caso de que sea un usuario sin token(anonymous) pues rechazarlo.... Ésto lo hago igualandolo a los tipos de permisos que queremos que tenga la vista en base a la autenticación `..[IsAuthenticated]`

Postman para realizar consultas incluyendo Token

Se instaló el Postman para realizar las respectivas consultas

Permission Is Super User

Se realizó la prueba en el Postman mandando en los headers el token de cada usuario y se recuperó efectivamente los productos de ése usuario

Dejemos claro Permission y Authenticate

Se plantea como requerimiento listar todos los productos que tienen stock

Se hace la vista, el manager, se usa el mismo serializador. Pero se hace énfasis en la diferencia entre el `permission` y el `authentication`

el `authentication` solo VERIFICA que el token existe nada más(no interfiere en las líneas de código) en tanto que el `authentication` habilita la autenticación ya sea por si es Admin, por token, etc Por ejemplo, si tengo un usuario que no está autenticado por token:

```
class ListProductoStock(ListAPIView):
    # Se lista todos los productos que tienen stock
    serializer_class = ProductSerializer
    authentication_classes = (TokenAuthentication,)
    #permission_classes = [IsAuthenticated, IsAdminUser]

    def get_queryset(self):
        return Product.objects.productos_con_stock()
```

Esto me muestra de igual manera la vista a pesar de que sea el usuario anonimo, ojo que sí verifica que exista un token pero no impide el flujo de otras líneas de código

```
class ListProductoStock(ListAPIView):
    # Se lista todos los productos que tienen stock
    serializer_class = ProductSerializer
    authentication_classes = (TokenAuthentication,)
    permission_classes = [IsAuthenticated] #Doy

    def get_queryset(self):
```

Esto no me muestra la vista puesto que si está habilitada la autenticación

```
class ListProductoStock(ListAPIView):
    # Se lista todos los productos que tienen stock
    serializer_class = ProductSerializer
    # authentication_classes = (TokenAuthentication,) #Esto u
    permission_classes = [IsAuthenticated, IsAdminUser] #Do

    def get_queryset(self):

        return Product.objects.productos_con_stock()
```

Otro caso :

Esto me muestra la vista Sí, pero aquí la autenticación está hecha directa en la consola del admin

Parámetros por URL Django Rest Framework

Se plantea como requerimiento hacer un filtro de productos dependiendo el género

Para ello pues se ha creado el respectivo manager, view, url y un serializador para los colores pero para yo poder hacer el filtro por productos

¿cómo mando/recibo el parámetro del género en la URL en DRF?

pues es de la misma manera que en Django normal solo mando el pk en la url que en este caso seria <gender> y luego lo recupero en mi vista con el kwargs

Recuperar valores de Peticiones GET- query_params

Lo que se hará es ver como mandar varios parámetros, para ello es similar a cómo se trabaja en Django normal. Lo único que varía es que en las vistas hago uso de el

query_params.get a diferencia del Django normal donde se usaba el GET.get

```
nombre = self.request.query_params.get('name',None)
```

Json dentro de Json - Reporde de Ventas - Caso práctico

Haciendo referencia a la app ventas pues se tienen 2 modelos que hacen referencia a factura cabecera y factura detalle.

Entonces se plantea como requerimiento que mostremos un reporte de ventas pero para cada venta mostrar la lista de productos

Si bien se puede solucionar el requerimiento cambiando el serializador de mi modelo a fin de que liste todos los detalles pues aquí se lo hará de una forma mucho más sencilla

De momento únicamente estoy listando las ventas sin el detalle en la siguiente clase se listará con el detalle

Serializador dentro de un MethodFieldSerializer

Para poder obtener los detalles primeramente tendría que modificar mi serializador pues añadiría un methodserializer con el cual accedería por cada interacción de mi venta a su venta detalle respectiva

Dentro de mi método hiciera el respectivo query(que retorna productos) con ayuda de los managers en el cual mandaré como parámetro el id de mi venta cabecera

Ahora a ése query necesito que el detalle de venta me los pinte de manera correcta es decir a cada producto necesito que se serialize entonces creo otro serializador para estos productos y

ya volviendo al método creo otra variable a la cual le igualo mi serializador y le paso como parámetro el query, como son muchos el many=True y al final el .data para para acceder al serializador

```
def get_productos(self, obj):
    query = SaleDetail.objects.productos_por_venta(obj.id) # Aquí obtengo mi lista de productos
    #Aquí a la lista de productos la serializo con mi serializador pero para acceder en sí a mi serializador debo poner el .data al final
    productos_serializados = DetalleVentaProductoSerializer(query, many = True).data
    return productos_serializados

class DetalleVentaProductoSerializer(serializers.ModelSerializer):
    class Meta:
        model = SaleDetail
        fields = (
            'id',
            'sale',
            'product',
            'count',
            'price_purchase',
            'price_sale',
        )
```

Serializador Dentro de Serializador - Proceso Venta

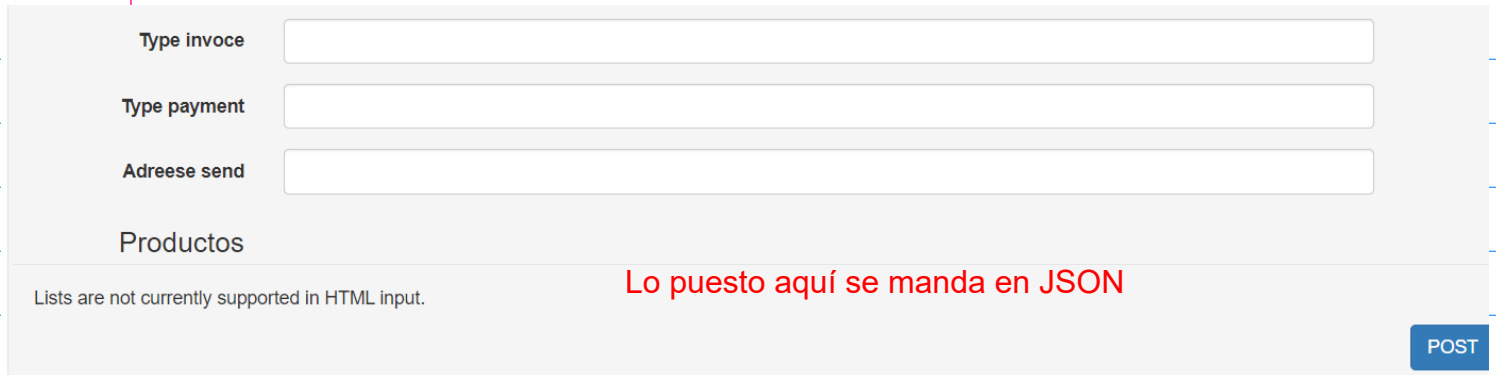
En éste caso lo que haremos es registrar una venta, tomar en cuenta que para crear una venta se trae una lista de productos o el detalle , primero se guarda los datos de la venta y luego se manda el detalle Entonces, ¿bajo qué modelo se trabaja?

Primero crearemos un serializador para la venta, no llamaremos a todos los datos puesto que los mismos se calculan, la venta va a tener productos/detalles y estos productos voy a tener que listarlos entonces de igual manera voy a necesitar otro serializador para productos donde no voy a poner estrictamente todos los datos mas bien solo los que necesite y no se calculen

Método Create en CreateAPIView - Proceso Venta

Para poder crear la venta trabajaremos con el CreateAPIView por ende crearemos una vista, en la cual haremos uso del serializador. De igual forma crearemos una url para la vista...

Sin embargo se puede observar que no me dibuja correctamente para los productos, entonces debemos hacerlo manualmente en el Postman lo cual lo haremos posteriormente.



Type invoice

Type payment

Adreese send

Productos

Lists are not currently supported in HTML input.

Lo puesto aquí se manda en JSON

POST

Ahora tomando en cuenta el hecho de que los serializadores los estamos haciendo no relacionados con el modelo pues lo que tenemos que hacer es métodos para poder por ejemplo guardar una instancia.

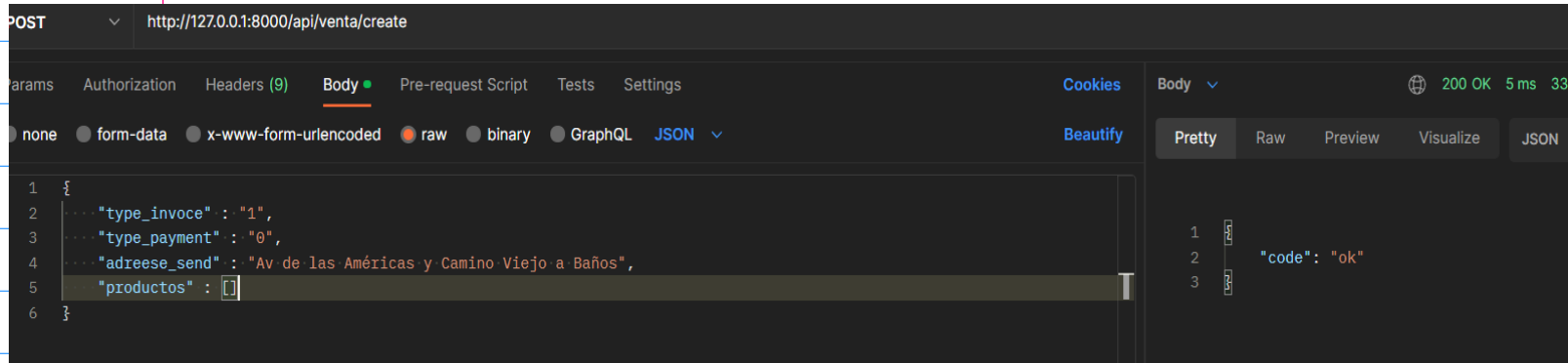
Para ello sobreescribiremos una función la cual es la create en la cual deserializaremos el JSON que se está mandando, posteriormente lo valido y al final lo recupero

```
def create(self, request, *args, **kwargs): #función a sobreescribir
    serializer = ProcesoVentaSerializer(data=request.data) #aquí deserializ
    #
    # Ahora debemos validar si la información es válida
    serializer.is_valid(raise_exception = True) # con el raise indico que me
    #Ahora recupero la info que nos está mandando
    tipo_recibo = serializer.validated_data['type_invoice']
    print('*****', tipo_recibo)
    return None
```

Validated_data de un Serializador

Recupera datos de un serializador

Aquí se hizo las pruebas respectivas con el postman y la corrección de que siempre un CreateAPIView necesita un response como retorno



```
***** 1
[03/Oct/2022 16:26:37] "POST /api/venta/create HTTP/1.1" 200 13
Quokka
```

Recibir un atributo de tipo colección de objetos en un serializador

Entonces ahora voy a crear el objeto venta con los datos que me mandan, ojo que los que no se ponen ya tienen un valor por default o no son obligatorios

```
venta = Sale.objects.create(
    date_sale = timezone.now(),
    amount = 0,
    count = 0,
    type_invoice = serializer.validated_data['type_invoice'],
    type_payment = serializer.validated_data['type_payment'],
    adrese_send = serializer.validated_data['adrese_send'],
    user = self.request.user,
    # Los demás datos se están poniendo por default
)
# Recuperamos los productos de la venta
```

Entonces además de esto estoy mandando mis productos en base a como mi serializador está construido

```
# Los demás datos se están poniendo por default
)
# Recuperamos los productos de la venta
productos = serializer.validated_data['productos']
print('&&&&& ', productos)
return Response({'code': 'ok'}) #Ojo que siempre un
```

```
class ProcesoVentaSerializer(serializers.Serializer):
    type_invoice = serializers.CharField()
    type_payment = serializers.CharField()
    adrese_send = serializers.CharField()
    productos = ProductDetailSerializers(many=True)
```

```

{
  "type_invoice": "1",
  "type_payment": "0",
  "adreese_send": "Av de las oblatas y Camino Viejo a cañar",
  "productos": [{"pk": 2, "count": 2}, {"pk": 3, "count": 1}]
}

```

```

1
2
3
{
  "code": "ok"
}

```

Pero ojo que únicamente estoy creando mi cabecera o mi venta, mi detalle aún no lo estoy creando sin embargo, ya estoy recuperando los productos

Registro detalle venta con bulk create

En éste caso para no registrar a cada rato un registro de producto lo haremos de golpe y para ello usaremos el bulkcreate que ya vimos en la sección de modelos avanzados con el cual crearemos de una sola todos mis productos y no uno por uno..

Entonces de los productos que mando desde el postman en JSON pues los itero a cada uno y de cada uno obtengo su ID, lo busco y lo asigno a una variable

Entonces con ésto ya puedo crear el detalle pero Ojo que para crear el detalle estoy mandando atributos propios del producto que recupero, es decir usando el modelo en tanto que para la cantidad por ejemplo estoy usando el serializador puesto que es un dato que me facilita el cliente y no es algo que se recupera del modelo

```

ventas_detalle = []
#
for producto in productos:
    prod = Product.objects.get(id=producto['pk']) #Ojo que aquí va pk debido a que en el s,

    venta_detalle = SaleDetail( # Creo mi detalle de venta
        sale = venta,
        product = prod,
        count = producto['count'], # atributo del serializador y del modelo
        price_purchase = prod.price_purchase, # atributo propio del modelo
        price_sale = prod.price_sale,
    )
    amount = amount + prod.price_sale * producto['count'] # saco el monto toal
    count = count + producto['count'] # saco la cantidad
    #Añado mi detalle a mi lista de detalles
    ventas_detalle.append(venta_detalle)

venta.amount = amount
venta.count = count # Cantidad de Productos
venta.save()

#
SaleDetail.objects.bulk_create(ventas_detalle)
return Response({'mensaje': 'Venta exitosa'}) #Ojo que siempre un CreateAPIView necesita u

```

Una vez hecho eso añado mi detalle a mi lista de detalles,
 Actualizo el monto de la venta
 Actualizo la cantidad de productos en la venta
 Guardo mi venta
 Creo mi venta detalle

ListField Serializer - Serializador para campos de tipo array

Lo que se plantea es tratar de optimizar lo que sería el bucle for que por cada iteración ya que cada iteración busca un producto con el mismo pk...

Para solucionar esto lo que debemos hacer es que

Similar a como antes con el bulkcreate hicimos un guardado de golpe pues también podemos hacer un RECUPERADO DE GOLPE, sin embargo debemos modificar nuestro serializador. Para ello haremos uso de esto

```
prod = Product.objects.filter(id__in=[1,2,3,4,5,6])
```

el in lo que hace es esperar un array en el cual estaran todos los objetos con el id que se indique..

Entonces para yo poder hacer uso de éste filter con in pues debo modificar mi serializador, pero en éste caso crearemos otro

Entonces para los productos en éste caso recibiremos un array y para ello usaremos el `ListField`, sin embargo debemos previamente definir otro serializador que indique de qué tipo va ser el array, éste serializador hereda de `ListField`

Dentro de mi serializador especificamos de qué tipo quiero que sea la lista y luego ya volviendo a mi `Serializador inicial` pues igualos los productos a este `Serializador`

¿Pero en este caso como especifico las cantidades de cada producto? en el anterior serializador pues lo hacía mandando otro serializador dentro del mismo, sin embargo aquí lo hago simplemente añadiendo otro atributo al serializador igualandolo de igual manera al serializador de arrays

```
class ProcesoVentaSerializer2(serializers.Serializer):  
  
    type_invoice = serializers.CharField()  
    type_payment = serializers.CharField()  
    adrese_send = serializers.CharField()  
    productos = ArrayIntegerSerializer()  
    cantidades = ArrayIntegerSerializer()
```

Registrar Venta utilizando ListField Serializer - Proceso venta Optimizado

Hemos creado otra vista en la que a diferencia de como recuperábamos los productos con el pk pues ahora lo que recuperaremos es un array con los id de cada producto, es lo hacemos como se dijo anteriormente con el id__in

Sumado a eso para las cantidades de igual manera se recupera llamando al serializador y ya no con el modelo

y

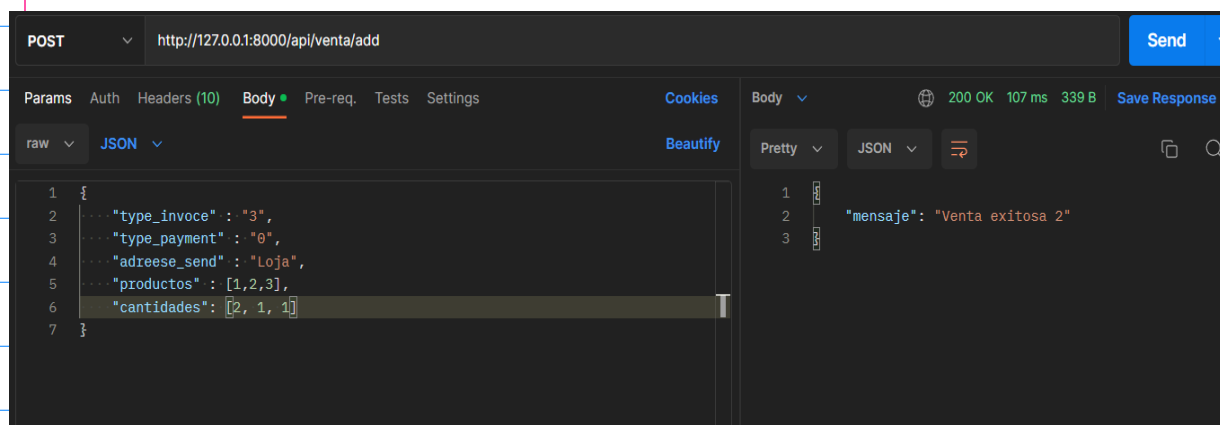
para poder iterar dos listas a la vez en python hago uso de zip logrando asignando la cantidad al producto respectivo

y al final añado las url normalmente

```
# variables para venta
amount = 0 # monto total de venta
count = 0
# Recuperamos los productos de la venta
productos = Product.objects.filter(
    id__in= serializer.validated_data['productos'] #De esta forma recupero un array con id de los productos
)
#
cantidades = serializer.validated_data['cantidades']
#
ventas_detalle = []
#
for producto, cantidad in zip(productos, cantidades):
    venta_detalle = SaleDetail( # Creo mi detalle de venta
        sale = venta,
        product = producto,
        count = cantidad, #gracias a que puedo iterar dos listas pues logro asignar la cantidad de un producto
        price_purchase = producto.price_purchase, # atributo propio del modelo
        price_sale = producto.price_sale,
    )
    amount= amount + producto.price_sale * cantidad # saco el monto toatl
    count = count + cantidad # saco la cantidad
    #Añado mi detalle a mi lista de detalles
    ventas_detalle.append(venta_detalle)

venta.amount = amount
venta.count = count # Cantidad de Productos
venta.save()

#
SaleDetail.objects.bulk_create(ventas_detalle)
return Response({'mensaje': 'Venta exitosa 2'}) #Ojo que siempre un CreateAPIView necesita un response como re
```



CRUD con ViewSet para Colors

Un ViewSet sigue siendo de igual manera una vista que funciona como todas las vistas de DRF. Sin embargo, tiene una diferencia clave, que el View Set resume el código aún mucho más.

Se hará un ejemplo bastante básico que va a tratar sobre un CRUD para el modelo COLORS, se seguirá usando un serializador, authentication_class y permission_classes.

```
class ColorViewSet(viewsets.ModelViewSet): #Hago uso del ModelViewSet debido a que lo relaciono directo con el modelo
    serializer_class = ColorsSerializer
    # Los ViewSets siempre para inicializarlos nos piden un parámetro, el queryset
    queryset = Colors.objects.all()
```

Todo esto ya es el CRUD

Viewsets.py

Ahora para hacer lo que sería la URL pues sí varía bastante ya que se hace uso de los ROUTERS

- Los archivos viewsets.py y routers.py son creados a voluntad a fin de que sea mas ordenado
- El router internamente ya trae procesos y maneja de cierta forma las peticiones pues ya detecta si las mismas son de tipo GET, POST, etc.

```
from rest_framework.routers import DefaultRouter
#
from . import viewsets

router = DefaultRouter()
```

De momento en el routers.py solo creamos una instancia del router, ya en la siguiente clase se utilizará.

GET Y POST en un Router

Lo que haremos en nuestro archivo routers.py será registrar la ruta, para ello usamos el router.register en el cual como parametros mando:

1. r'path' de la ruta
2. viewset
3. nombre (en este caso se debe poner basename)

Luego al igual que como se hacía con el urls.py pues defino un urlpatterns al cual le igualare mis ruta de mi objeto router

```
router = DefaultRouter()

router.register(r'colors', viewsets.ColorViewSet, basename="colors") # r'colors' es lo que sería la ruta

urlpatterns = router.urls #Aquí llamo a las urls de mi objeto router
```

Una vez se accede a ésta vista en el path pues se puede observar que la misma nos muestra toda una lista de colores(GET)
y al final nos da una opción para poder añadir otro objeto por medio del POST

```
[{"color": "blanco"}, {"color": "Azul"}, {"color": "Magenta"}, {"color": "Beige"}]
```

Raw data

HTML form

Tag

POST

