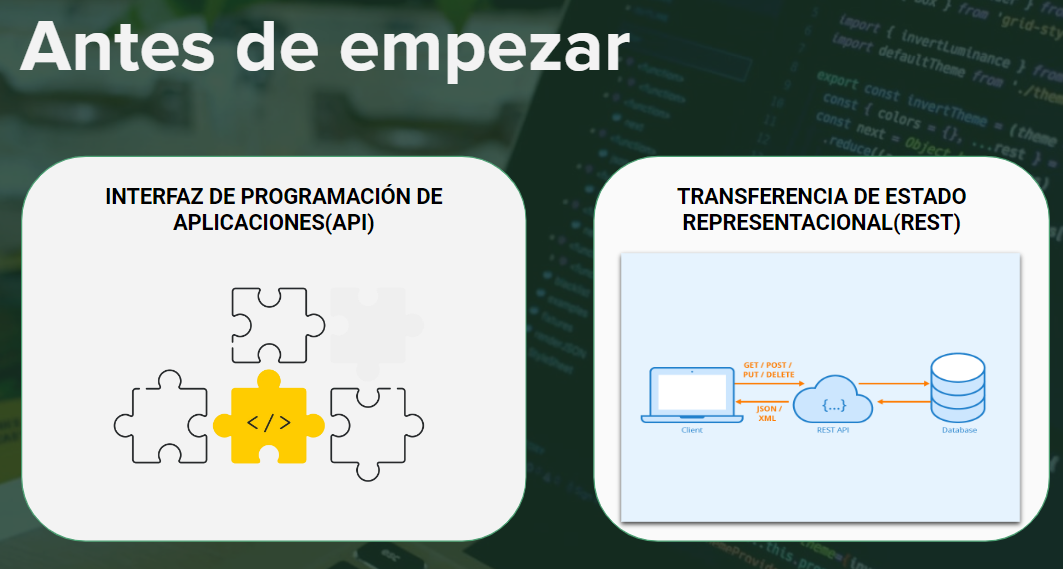
NOTAS CURSO **DJANGO REST FRAMEWORK** Developer.pe

<http://www.developerpe.com/>

Por: Oliver Sandoval

**Clase 1. INTRODUCCIÓN A DJANGO REST FRAMEWORK**

****

Una API (Application Programming Interface) es un conjunto de reglas y protocolos que permite la comunicación entre diferentes aplicaciones de software. Es un intermediario que permite que dos sistemas se comuniquen y compartan información de manera estructurada y estandarizada.

Las API facilitan la interacción entre aplicaciones al proporcionar un conjunto de funciones, métodos y endpoints que permiten el intercambio de datos de manera controlada. Estas pueden ser utilizadas tanto por desarrolladores internos como externos para acceder a servicios, funcionalidades o datos específicos de una aplicación o sistema.

REST (Representational State Transfer) es un estilo de arquitectura para el diseño de servicios web que se basa en estándares y principios simples. Se utiliza ampliamente para crear APIs que permiten la comunicación entre sistemas distribuidos.

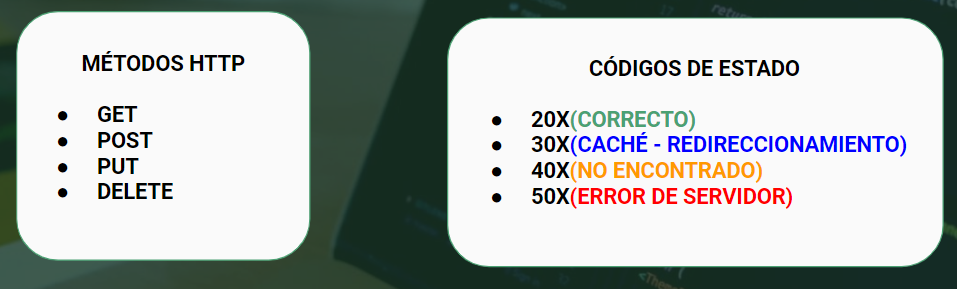
El funcionamiento de REST se basa en la comunicación cliente-servidor a través de solicitudes y respuestas HTTP. El cliente realiza una solicitud HTTP al servidor para acceder a un recurso, indicando el verbo HTTP correspondiente y, opcionalmente, datos adicionales en el cuerpo de la solicitud. El servidor procesa la solicitud y envía una respuesta HTTP al cliente, que puede incluir los datos solicitados o confirmar el éxito o fallo de la operación.

En resumen, REST es un estilo de arquitectura que proporciona principios y estándares para diseñar servicios web basados en recursos, operaciones CRUD y comunicación mediante solicitudes y respuestas HTTP. Permite la creación de APIs flexibles, escalables y de fácil consumo por parte de aplicaciones cliente.

La comunicación entre el cliente y el servidos se hace a través de JSON o XML:



Las peticiones en la comunicación se hacen a través de los métodos HTTP, y aquellas peticiones tienen un código de Estado:



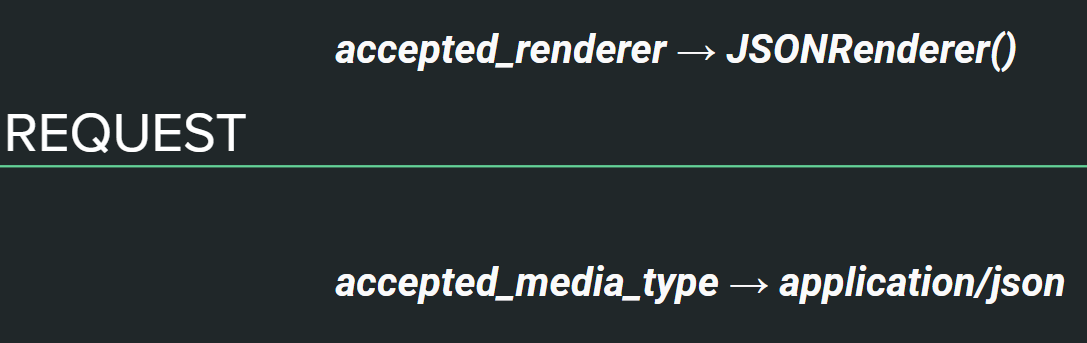
Para crear estas API REST se usa el siguiente framework:



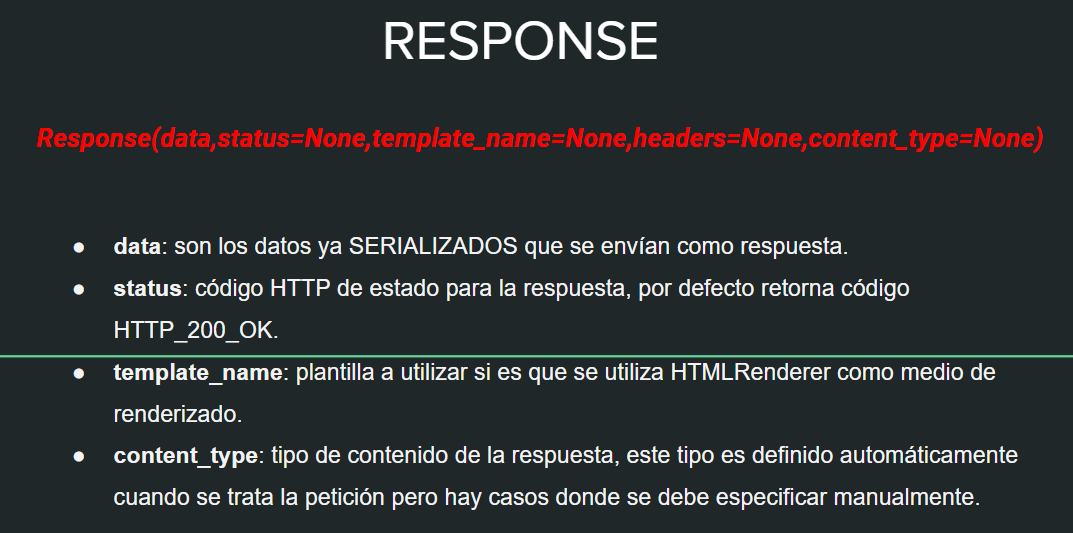
Diferencias entre Django y Django REST:



El REQUEST tendrá estos dos atributos:



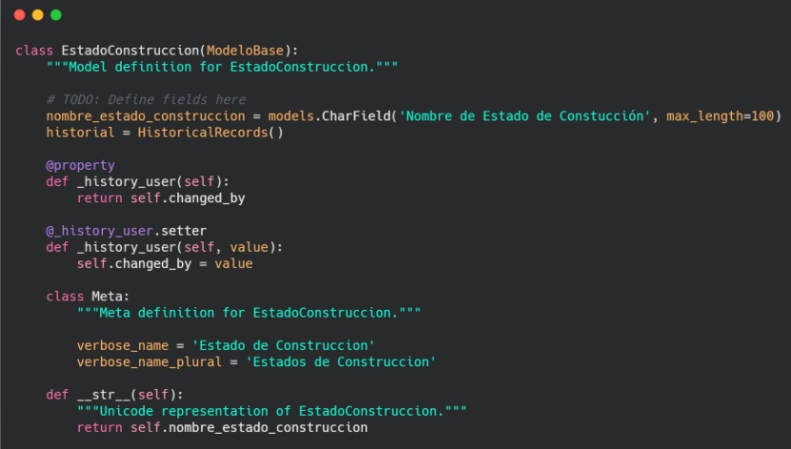
Y el RESPONSE los siguientes parámetros



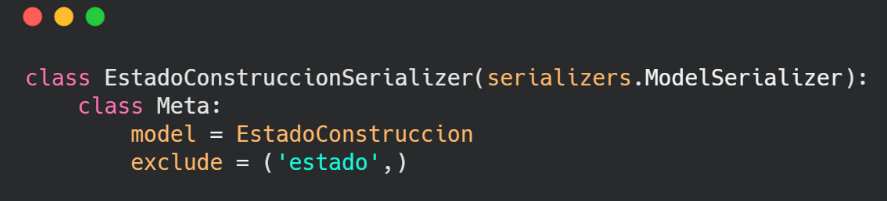
Proceso de desarrollo en DRF (Acá no existen los templates)



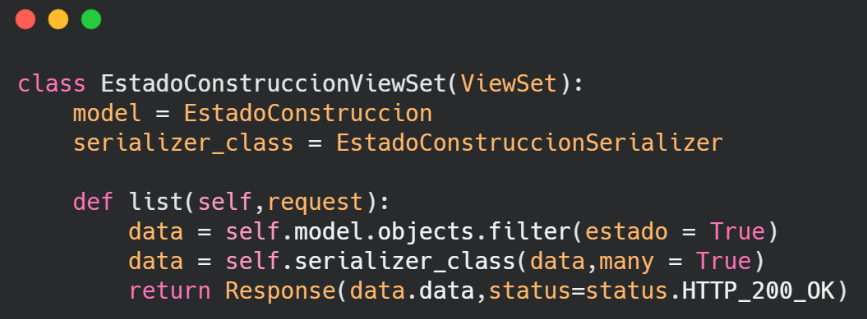
Veamos un ejemplo de modelo en DRF



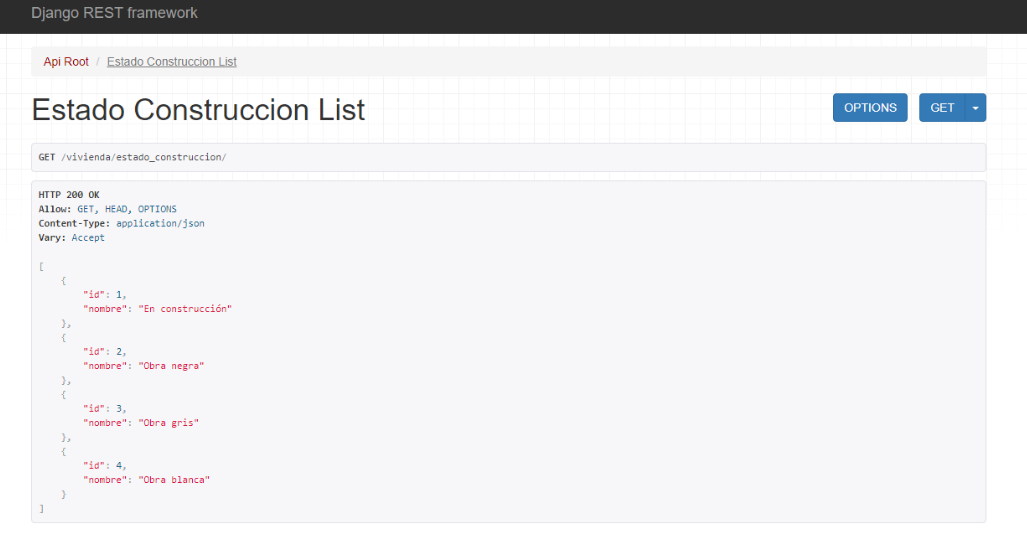
Luego un serializador



Ahora la vista



La ruta



**Clase 2. Instalación y configuración inicial de proyecto**

Creamos la carpeta en **Escritorio** del curso llamada **Django Rest,** dentro de ella creamos el entorno virtual llamado **django\_rest:**

* python -m venv django\_rest

y lo activamos:

* cd django\_rest\Scripts\activate

Ahora en la carpeta del curso instalamos **django y django rest** con el comando:

* **pip install djangorestframework**

y para crear el proyecto

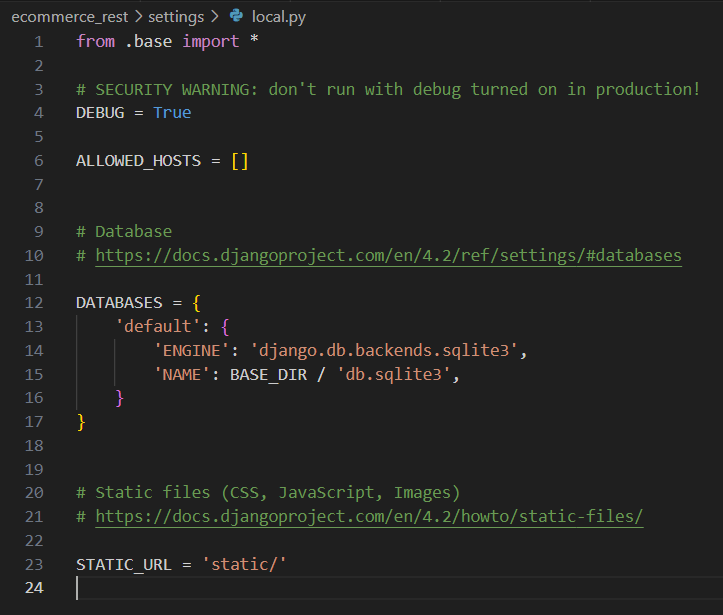
* **django-admin startproject ecommerce\_rest**

creándose la carpeta **escritorio/Djando Rest/ecommerce\_rest.** Dentro de **ecommerce\_rest** creamos la carpeta **settings** y en ella los archivos:

* \_\_init\_\_.py
* base.py
* local.py
* production.py

Todo el código de **settings.py** se pega en **base.py** con excepción de **DATABASES**, el cual va en **local.py y production.py.** Luego se elimina **settings.py.**

Los archivos **local.py y production.py** quedan iguales:



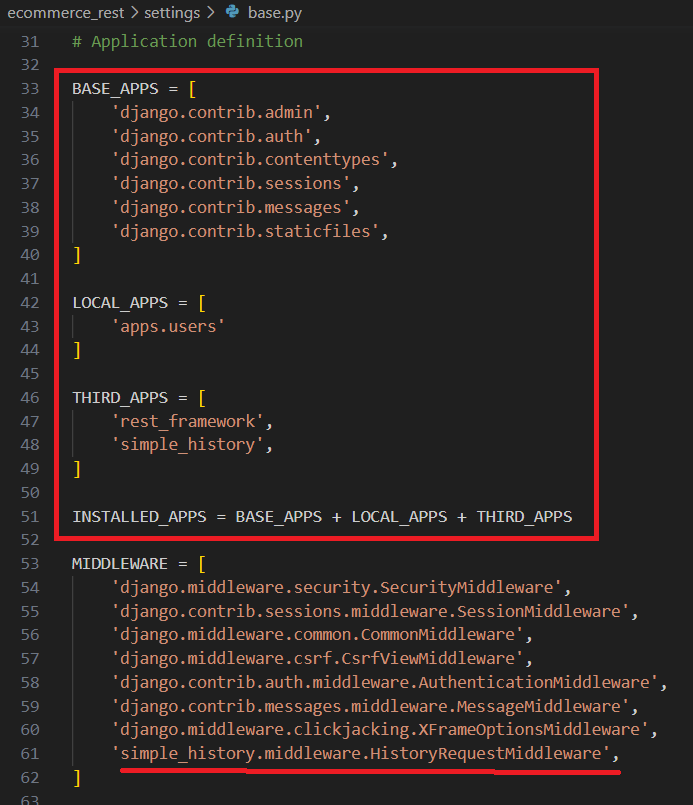
En **base.py** cambiamos el lenguaje:

* LANGUAGE\_CODE = 'es'

En **wsgi.py, manage.py y asgi.py** cambiamos la línea siguiente



En **base.py** cambiamos:



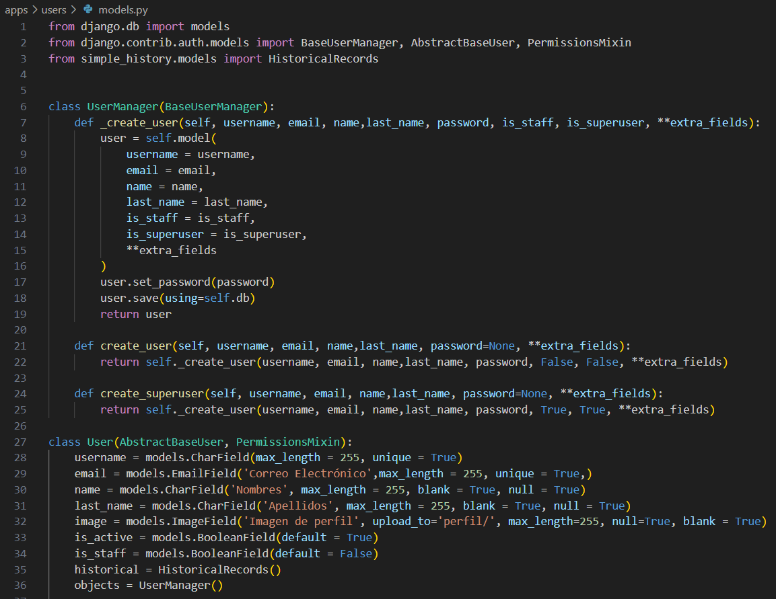
Instalamos dos por consola:

* **pip install django-simple-history**
* **pip install Pillow**

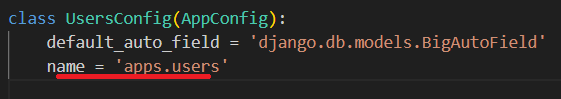
Creamos la carpeta **apps** dentro de **Escritorio\Django Rest\ecommerce\_rest,** luego desde **Escritorio\Django Rest\ecommerce\_rest\apps** se crear el archivo **\_\_init\_\_.py** y desde consola de acá se crea la app de usuario

* **django-admin startapp users**

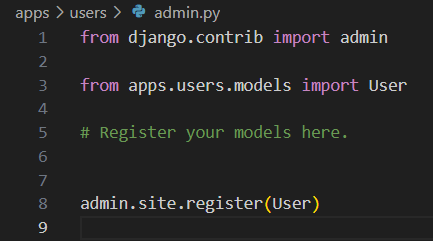
dentro de **models.py** de la app **users** pegamos los modelos de **github** del autor que están en la descripción del video.



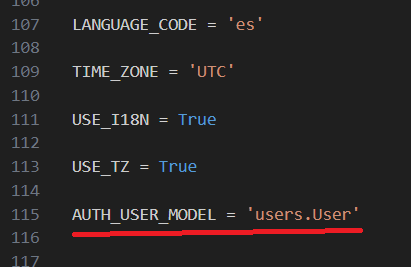
En **users/apps.py** corregimos la ruta de la app



En **users/admin.py** registramos la app users



Y en **base.py** agregamos



Ahora hacemos migraciones

* **py manage.py makemigrations**
* **py manage.py migrate**

Creamos superusuario

* **py manage.py createsuperuser**

con

* username: juan
* email: [juanm.96.andrade@gmail.com](mailto:juanm.96.andrade@gmail.com)
* nombres: juan Manuel
* apellidos: andrade rivera
* password: 12345

Corremos el proyecto

* **py manage.py runserver**

y entramos al Admin de django, verificando que todo funciona!

**Clase 3. Primer SERIALIZER y APIVIEW de usuario**

Un **serializador** es una clase que permite convertir objetos complejos, como modelos de Django, en tipos de datos nativos de Python y viceversa. Se utilizan para facilitar la serialización y deserialización de datos al interactuar con API RESTful.

La principal función de un serializador en Django REST Framework es:

1. **Serialización**: Convierte los objetos de Django en formatos de datos como JSON, XML o YAML, que son más fáciles de transmitir a través de una API. Esto implica tomar un objeto complejo, como un modelo de Django, y transformarlo en una estructura de datos más simple y legible.
2. **Deserialización**: Convierte los datos recibidos a través de una API en objetos de Django. Es decir, toma datos en formato JSON, XML u otro formato y los convierte en objetos de Django para su posterior manipulación en el código.

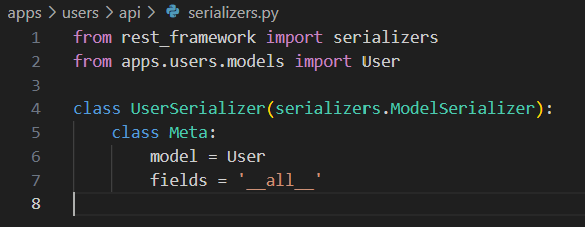
En cada **app** se crea una carpeta llamada **api,** luego creamos la carpeta **users/api,** dentro de ella los archivos:

* \_\_init\_\_.py
* serializers.py
* api.py
* urls.py

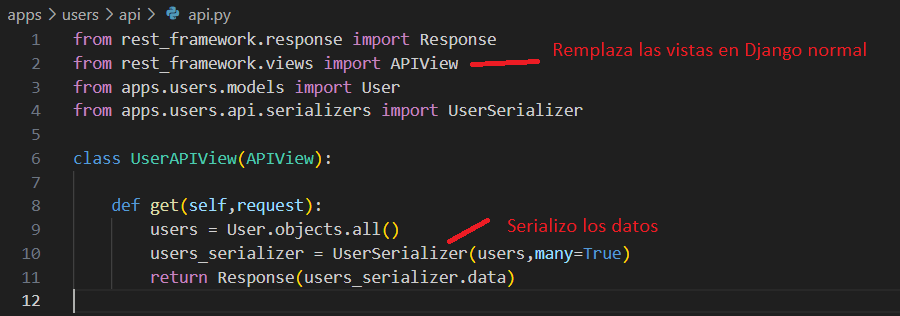
y borramos de **users**

* views.py
* test.py

Ahora creamos un **serializador** para el modelo **User** en **serializers.py.**

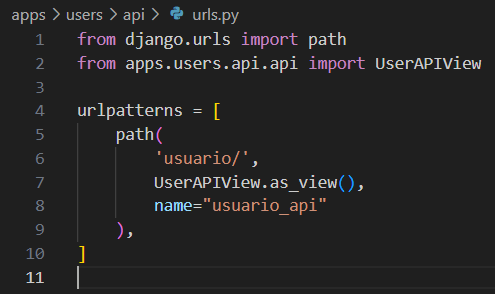


Luego creamos la **api** del **UserSerializer** creado en **api.py**:

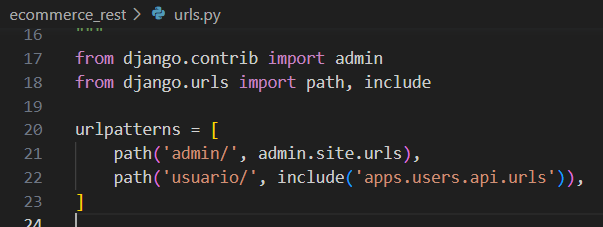


El método **get()** se utiliza para manejar las solicitudes HTTP GET a una API y es responsable de realizar las operaciones necesarias para recuperar los datos que se solicitan y luego devolver una respuesta HTTP que contenga estos datos solicitados por el cliente.

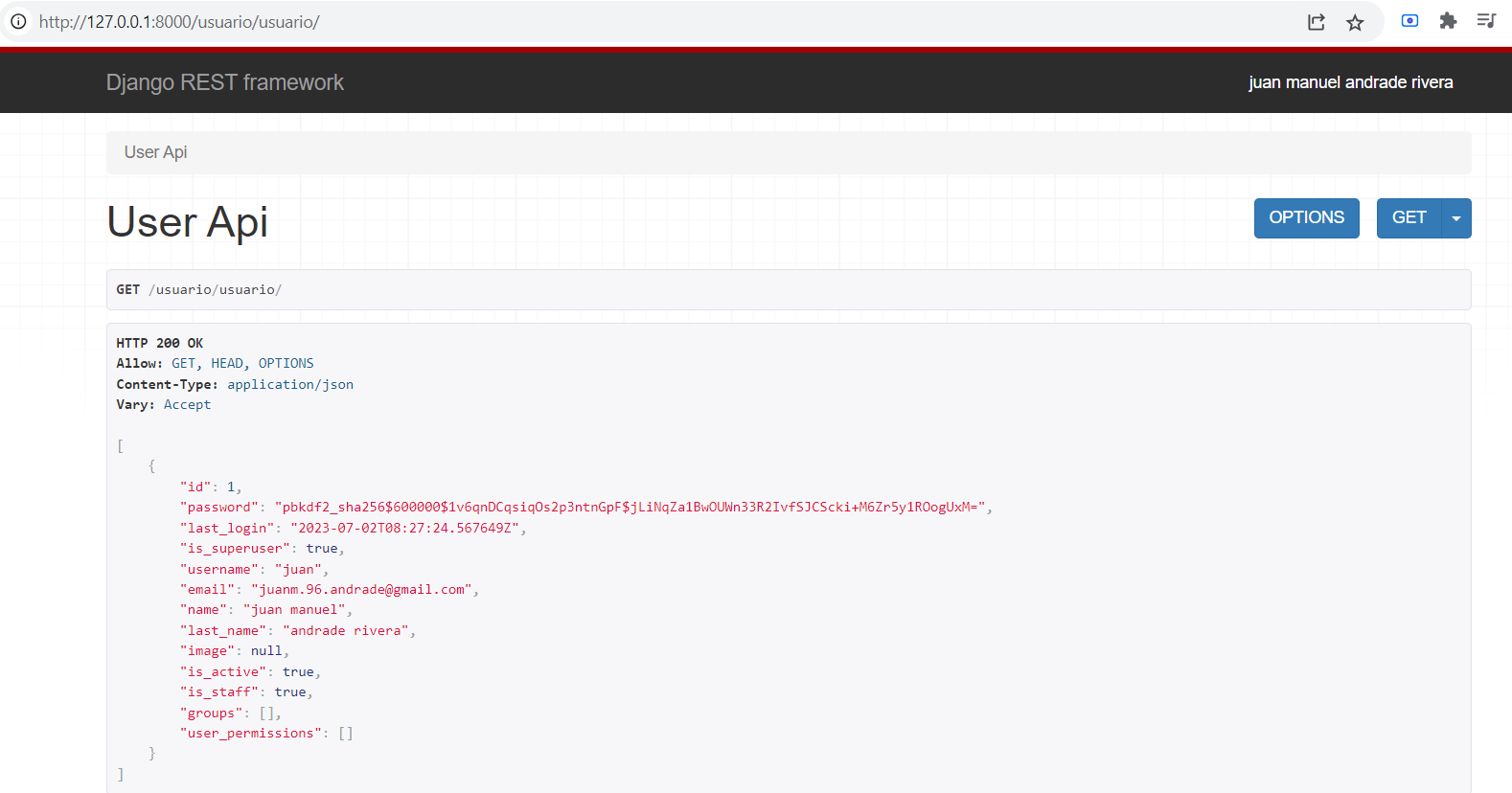
Ahora hay que enlazar las **urls** en **api/urls.py**

****

Con las **urls** del proyecto **ecommerce\_rest** en **urls.py**

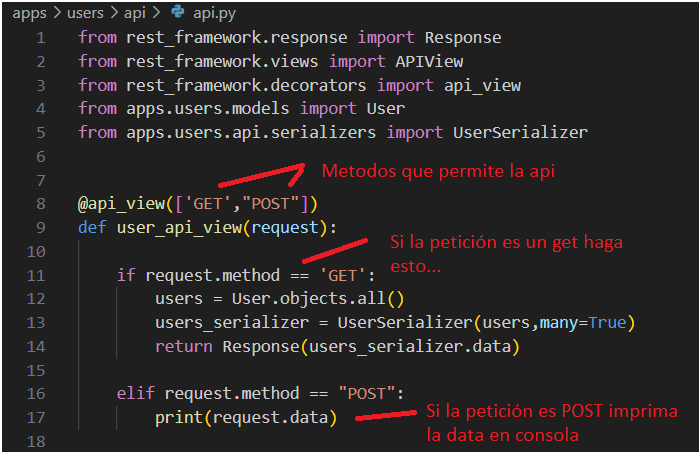
****

En el navegador veríamos:

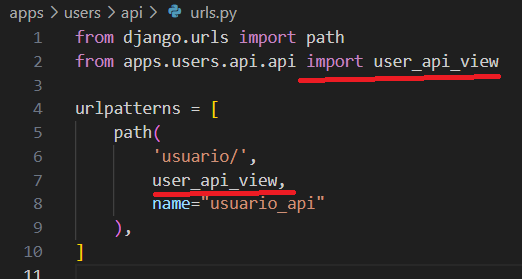


**Clase 4. Decorador @API\_VIEW**

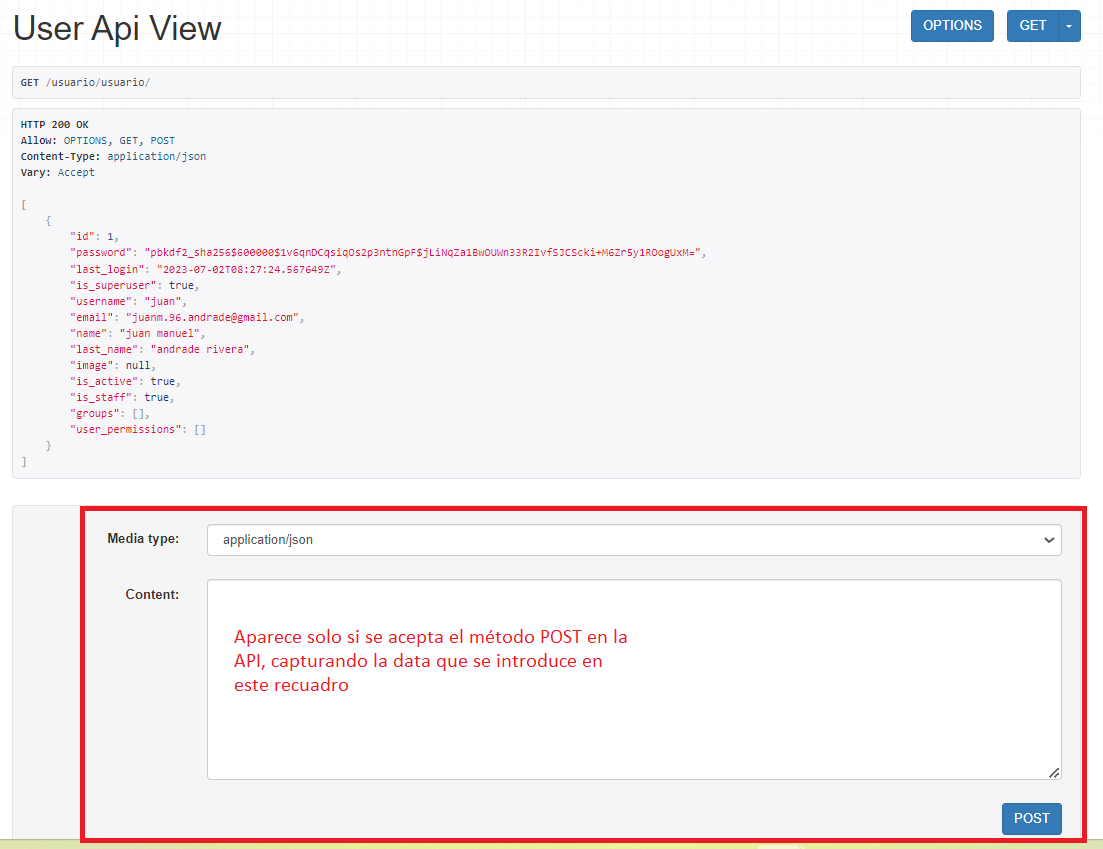
En esta clase haremos los mismo con funciones usando decoradores. Modificamos **api.py**

****

Corregimos la url asociada en **urls.py**

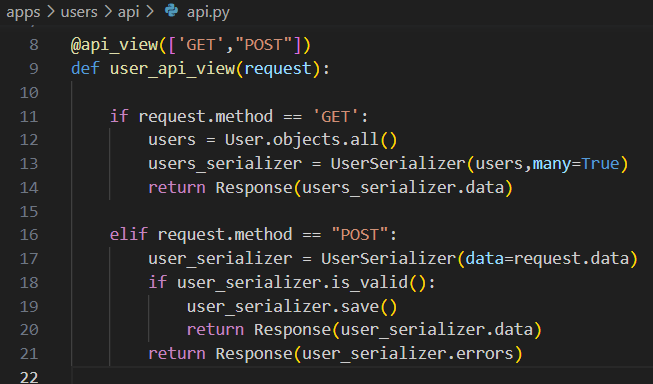


En el navegador veríamos



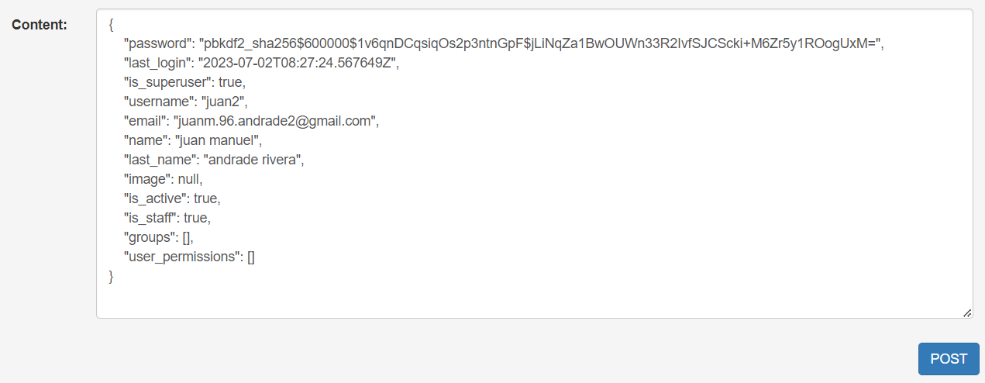
**Clase 5. Registro con @API\_VIEW y SERIALIZER**

Mejoremos el método **POST** para poder guardar un usuario que ingresa el usuario en formato **JSON**



Note que **users\_serializer = UserSerializer(users,many=True)** convierte en **JSON** el queryset que se consulta y luego se envia al usuario en formato **JSON.** Por otro lado, en el método **POST** el **user\_serializer = UserSerializer(data=request.data)** convierte el **JSON** en un queryset y si los datos empalman con el modelo entonces lo guarda, de lo contrario envia los errores de validación.

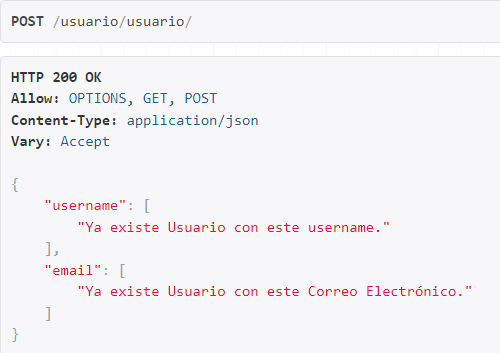
En el navegador al ingresar este nuevo usuario que cumple con los mismos campos del modelo usuario



Al darle **POST,** se captura y como es valido lo guarda y ahora tengo dos usuarios:



Pero si tengo un error en el **JSON** que envio, porque no cumple con alguna validación del modelo me aparece y no guarda el usuario



**Clase 6. DETALLE, UPDATE y DELETE con @api\_view**

Vamos a crear la api **user\_detail\_api\_view** para poder consultar (detalle – GET), actualizar (update - PUT) y eliminar (delete - DELETE) un usuario suministrando el **id** del usuario. Hacer el **CRUD.**

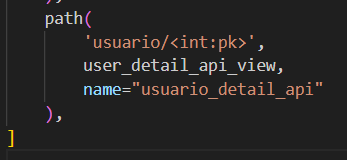


En la primera parte con **GET** obtenemos el usuario con su **pk.** Retornamos la consulta serializada con **user\_serializer.data.**

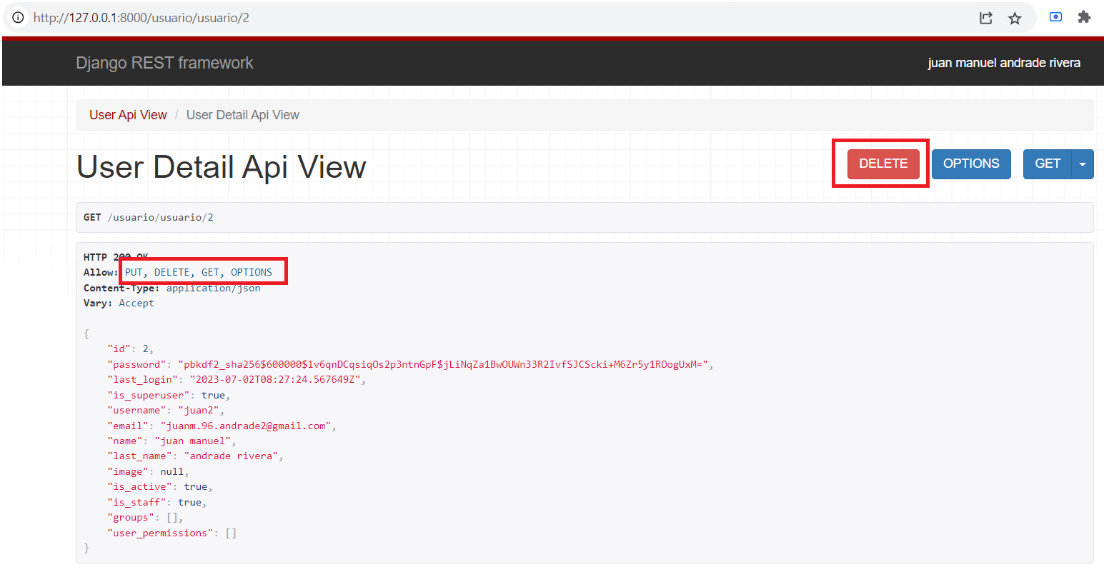
En la segunda parte con **PUT** consultamos el usuario en la variable **user,** luego con **UserSerializer(user,data=request.data)** cruza la data del usuario encontrado en la BD según el **id** y la data que ingresamos, luego si el cruce de información es valido entonces actualiza el registro, de lo contrario, retorna el error correspondiente.

En la tercera parte con **DELETE** elimina el usuario encontrado en la BD según el **id** y retorna el mensaje **“Eliminado”.**

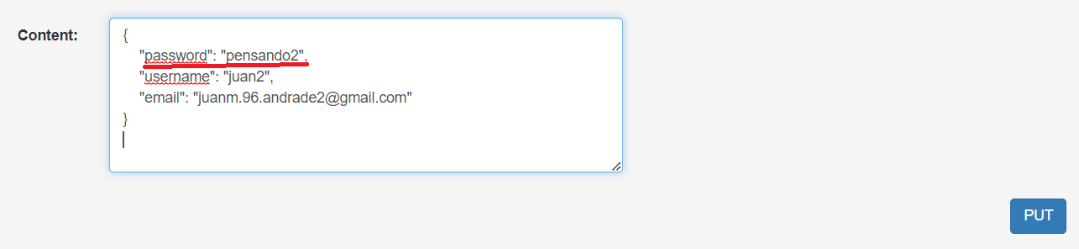
Ahora enlazamos su url, la cual recibe un número entero usada como **primary key** que se usa para hacer la consulta en la **apiview**



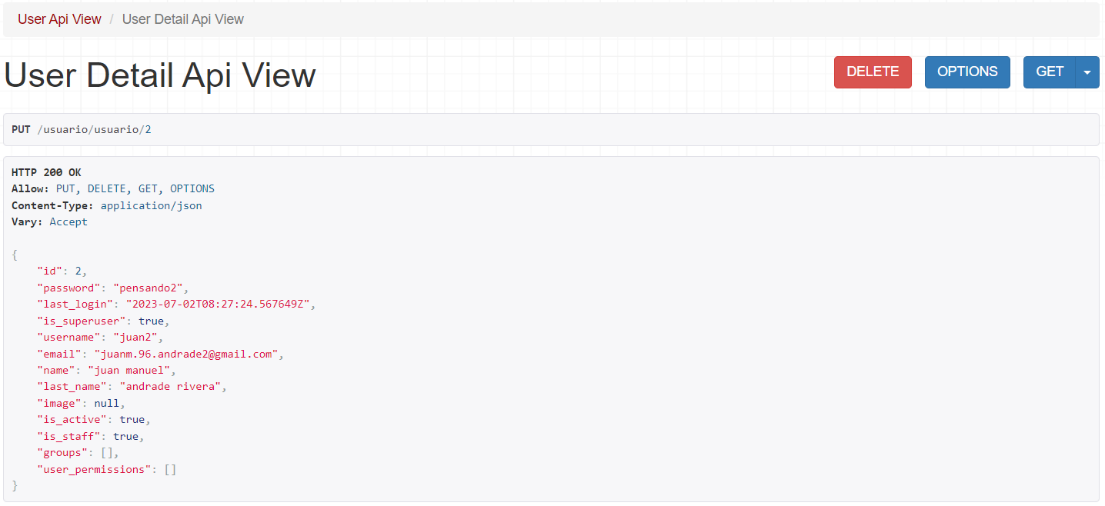
Para consultar el usuario con **id=2**



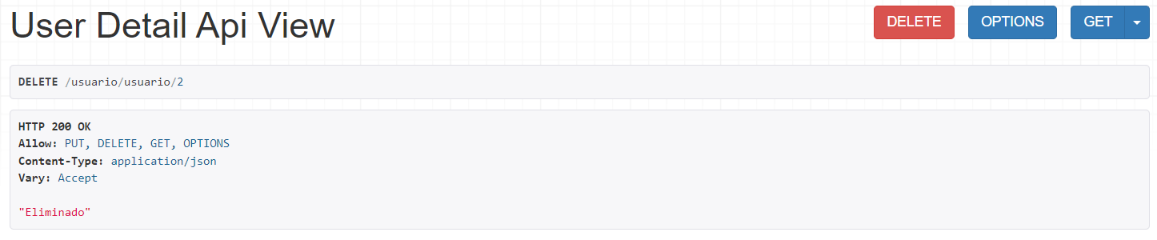
Para actualizar ese mismo usuario en el recuadro de abajo podemos actualizar los siguientes campos



Luego de dar **PUT** se actualizo la información.



Si quiero eliminar lo hago en el botón **delete** apareciendo



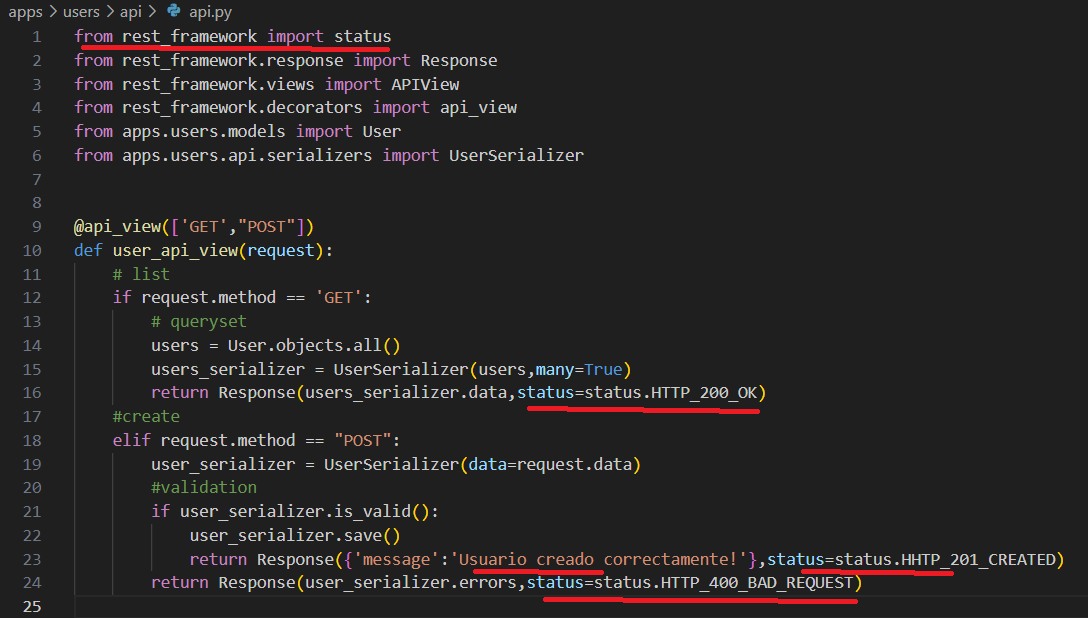
**Clase 7. Forma correcta de realizar un RESPONSE**

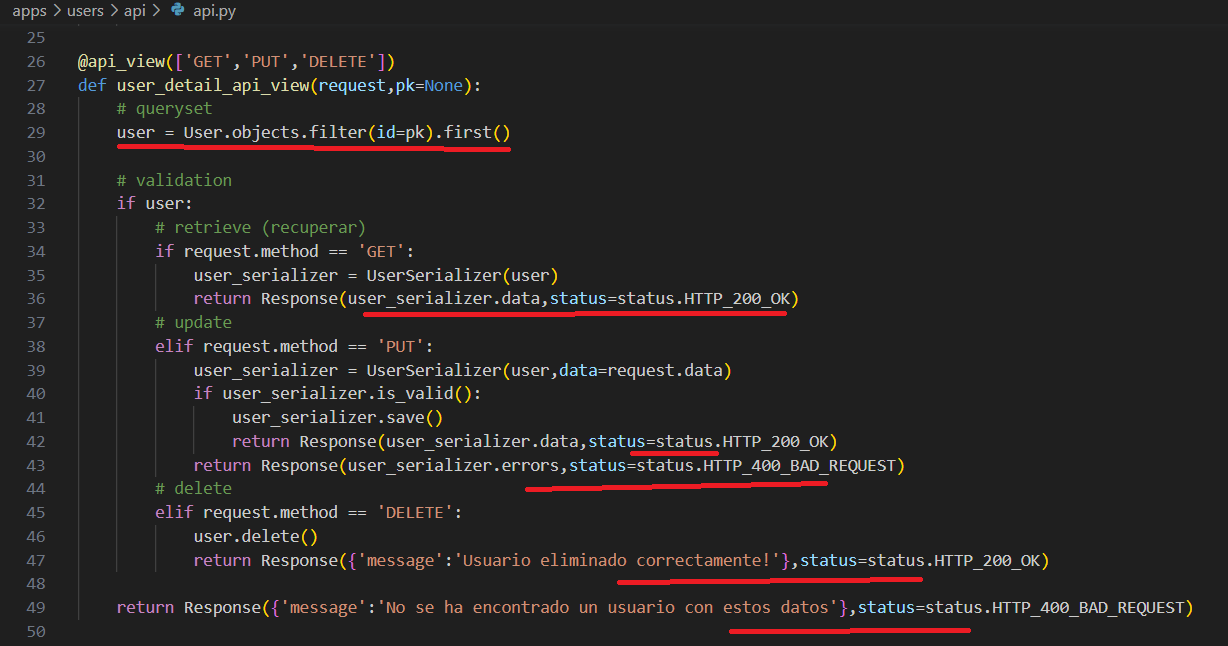
¡Hoy explicaremos la forma correcta de devolver un Response utilizando los **STATUS CODE!**

Aquellos status code (1xx – 2xx – 3xx – 4xx – 5xx) pueden ser consultados en

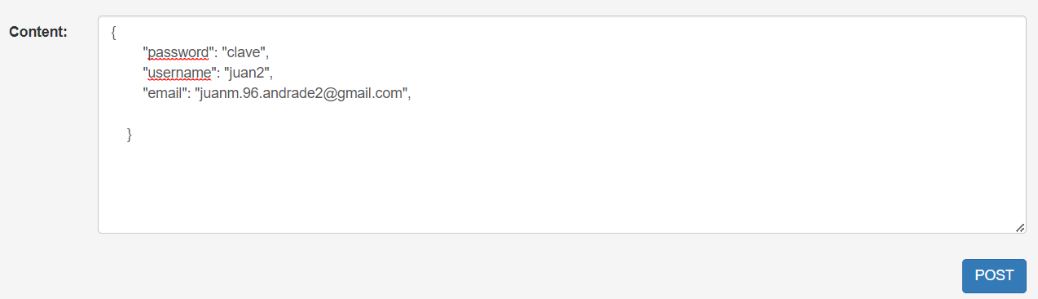
* <https://www.django-rest-framework.org/api-guide/status-codes/>

Lo anterior lo haremos mejorando un poco el código e introduciendo los **status** en cada **Response**:

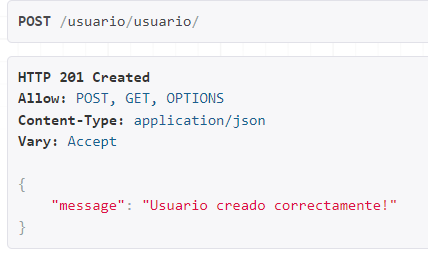




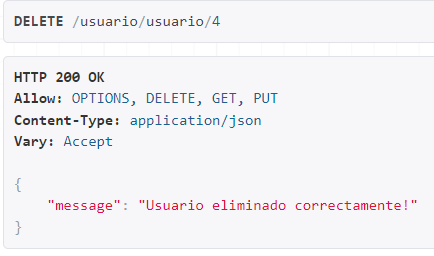
Si creamos este usuario



Nos sale luego

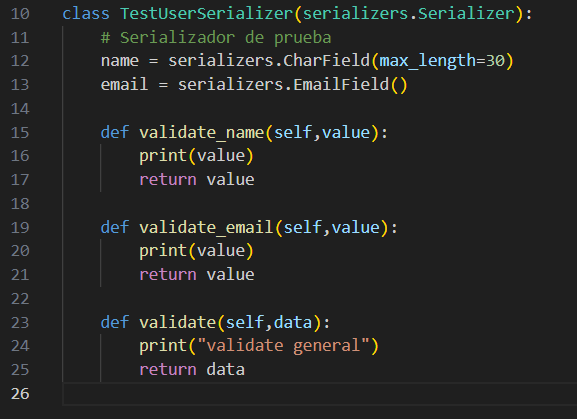


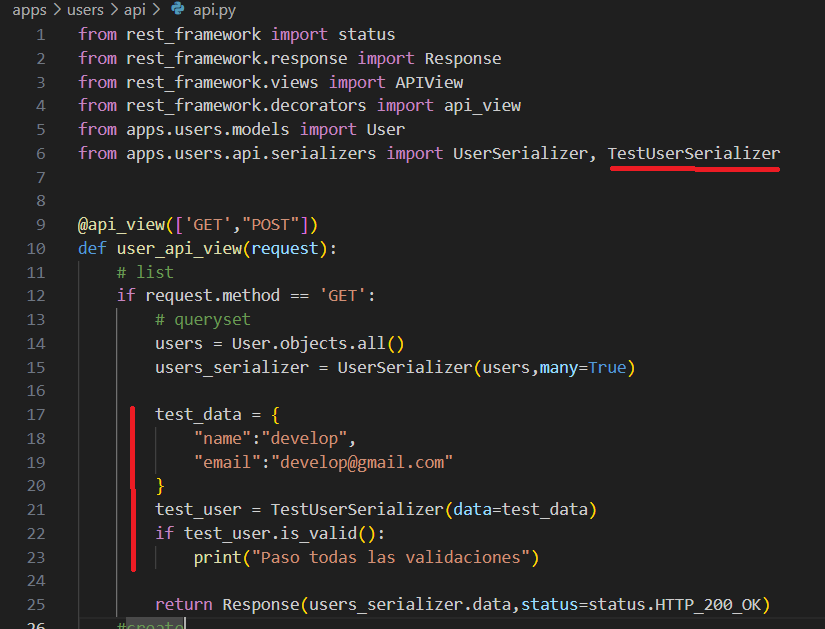
Si lo elimino



**Clase 8. Hablemos de los SERIALIZER**

Creamos un **serializer** de prueba para aprender de su funcionamiento y usamos la api ya creada anteriormente para probarlo:





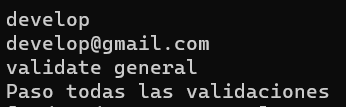
Note que los campos de los **serializadores** son los mismos que los **formularios** y los **modelos** en Django. Ahora el programa toma **test\_data** y lo serializa bajo el molde del **TestUserSerializer** esto bajo la línea

* test\_user = TestUserSerializer(data=test\_data)

luego valida si coincide con el molde serializador en

        if test\_user.is\_valid():

En el proceso de validación el valida si cada campo (uno a uno desde el primero al último) ingresado en **data** coincide con el formato pedido. Esto se observa en la consola al pasar por cada uno de los **validate\_name y validate\_email**



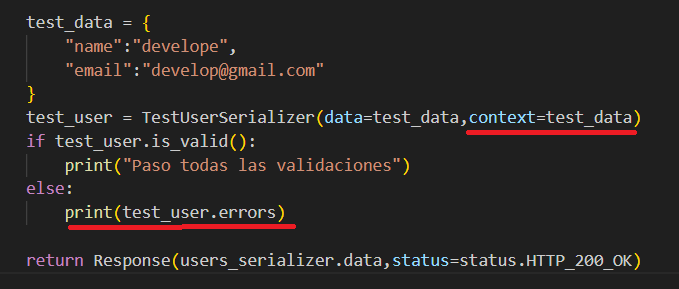
Si cada validación en cada campo esta bien, luego pasa al método **validate** y retorna la data serializada.

**Clase 9. Validaciones en un SERIALIZER**

Vamos a validar:

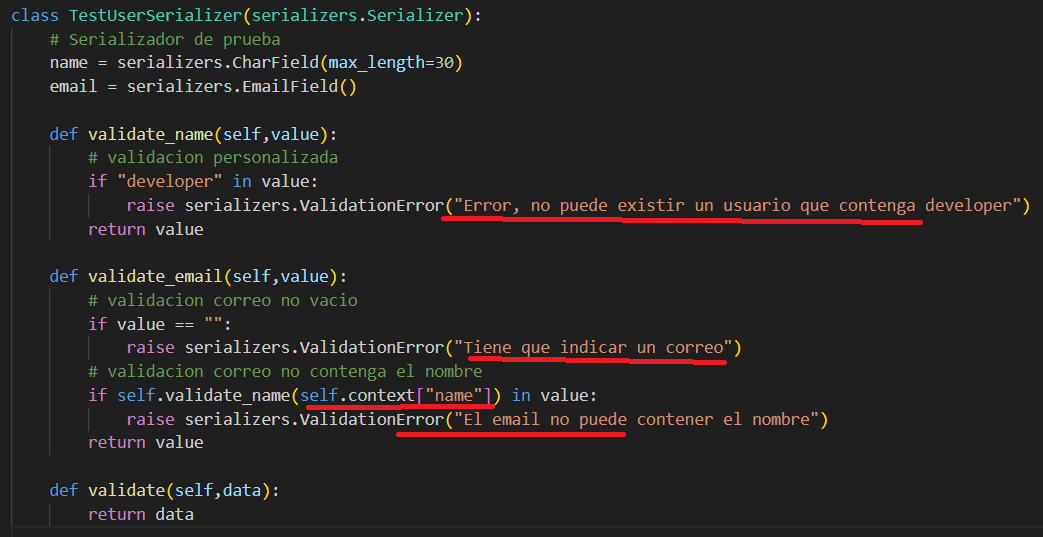
* Que el nombre no contenga la palabra “developer”
* Que el email no sea un espacio vacío
* Que el nombre no se encuentre dentro del email

Bajo la misma **api** note que



Usamos **context=test\_data** para luego poder manipular la data que ingresa de manera más fácil.

Ahora las validaciones en el **serializador** quedarían

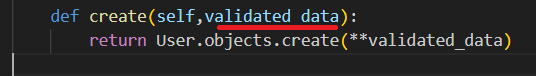


Si fallan las validaciones aparece en la consola los correspondientes mensajes, de lo contrario aparecerá **“Paso todas las validaciones”.**

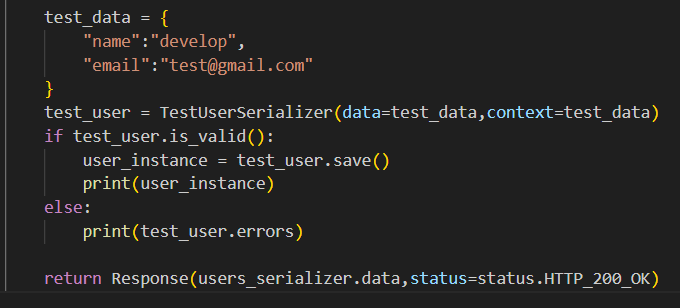
**Clase 10. Método create en un Serializer**

Veamos como el **serializer** crea un registro de usuario internamente.

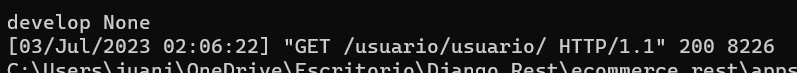
Agregamos el método **create()** debajo de los métodos de las validaciones



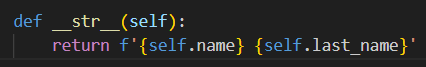
y la api quedaría



Lo anterior funciona así: primero se captura la información del usuario en **test\_user,** luego se valida con **test\_user.is\_valid()** e internamente esa **data** pasa a llamarse **validated\_data,** luego se guarda con **test\_user.save().** En este ultimo entonces el programa para hacer el **save** va al método **create()** en donde crea una instancia de la clase **User** con **User.objects.create(\*\*validated\_data)** y la muestra en consola dado el **print(user\_instance)**

****

Este nombre del registro se debe a que el modelo **User** representa sus instancias como



Pero en el Admin se puede visualizar

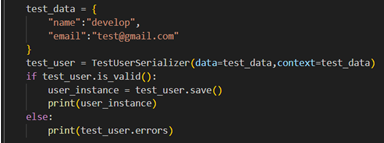


**Clase 11. Método UPDATE en un SERIALIZER**

Vamos a ver las actualizaciones usando **TestUserSerializer** con el método **PUT** para ver como ocurre.

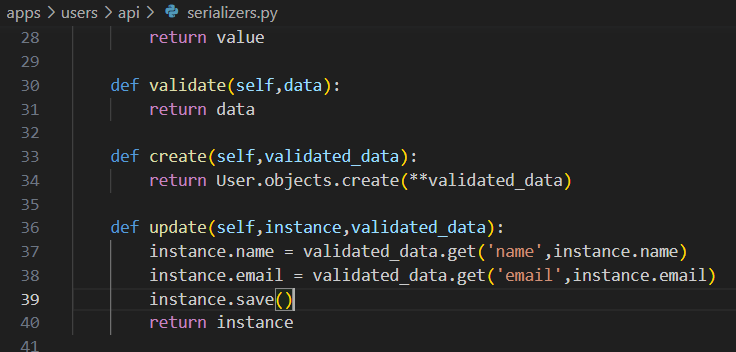
**Nota:** ¡Las validaciones se aplican para actualizar y para crear!

Eliminemos de **api.py**

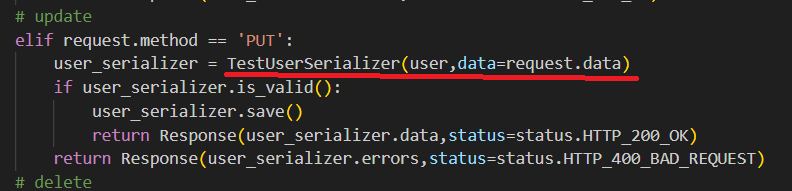
****

Y la validación de que el nombre se encuentre dentro del email.

Entonces plantemos el método **update()**

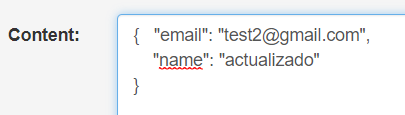
****

y el método **PUT** se modifica un poco

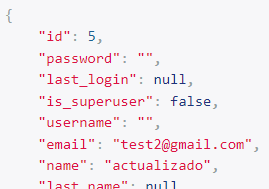


Entonces funciona así: primero la variable **user** guarda el registro de la instancia que ya existe en la BD, luego con **user\_serializer = TestUserSerializer(user,data=request.data)** se compara la instancia de la BD y la data ingresada por el método **PUT,** luego se validan los campos con **user\_serializer.is\_valid().** En caso afirmativo, el **user\_serializer.save()** hace referencia a aplicar el método **update(),** en donde campo a campo cruza la información y actualiza todo a una sola instancia, retornándola. El método instance**.save()** dentro del **serializer** hace referencia al método **save()** del modelo **User** dado que se aplica sobre una **instancia** de ese modelo**,** esto se ve mejor en el próximo video.

En el navegador actualizamos los datos con este input

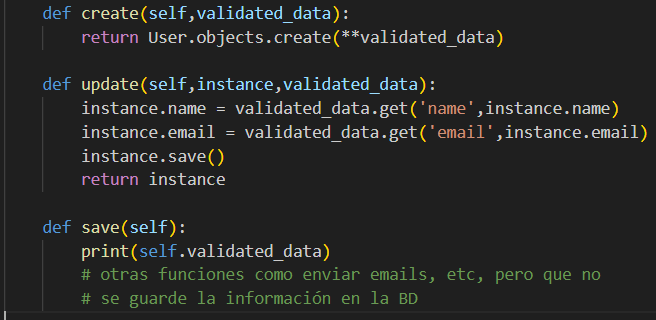


Y la nueva instancia queda



**Clase 12. Método SAVE en un Serializer**

Veamos el método **save()** en un **serializer**.

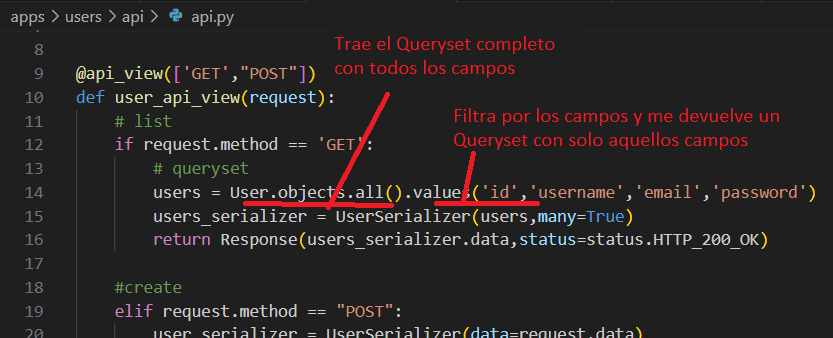
****

Este método **save()** omite los métodos **create y update** y realiza la lógica que se le codifique en él**,** si intentamos actualizar el registro anterior, no lo hace a pesar de no presentar errores.

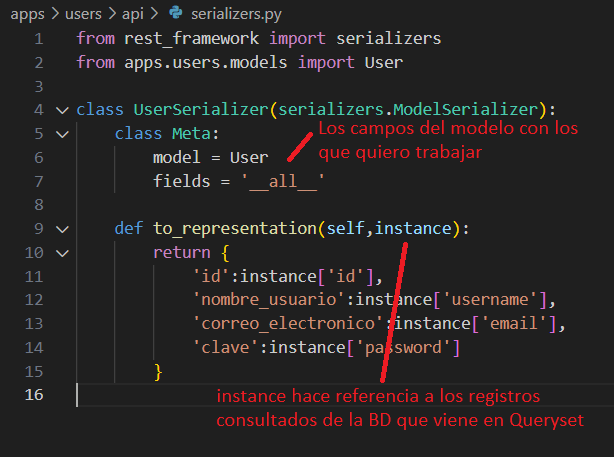
**Nota:** Este método es útil cuando quieres usar un serializador para realizar validaciones, pero no registros en la BD, por ejemplo, queremos que alguien se suscriba a nuestro blog y al llenar nuestro formulario de contacto, esta información se le envíe por correo pero no se me guarde aquel registro en la BD porque no tengo.

**Clase 13. To representation en un Serializer**

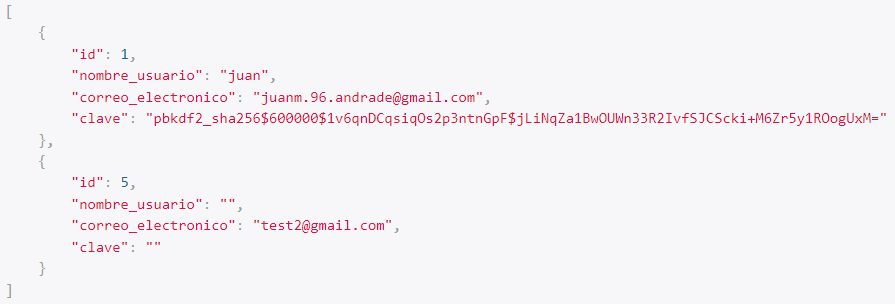
Borramos **TestUserSerializer** y trabajaremos solo **UserSerializer.** Vamos a mostrar el **id, username, email y password** de las instancias del modelo **User** que se consulten por el usuario**.** Para ello en **api.py** optimizamos la consulta del queryset para no traer todos los campos y los guardo en un queryset más pequeño con **.values(…)** en la variable **users**

****

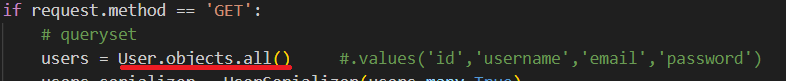
Vamos al **serializador** para definir qué información quiero presentar.

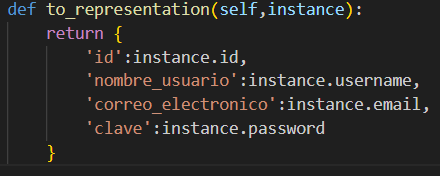


En el navegador veríamos las instancias del modelo de la siguiente manera:



**Nota:** podríamos haber hecho

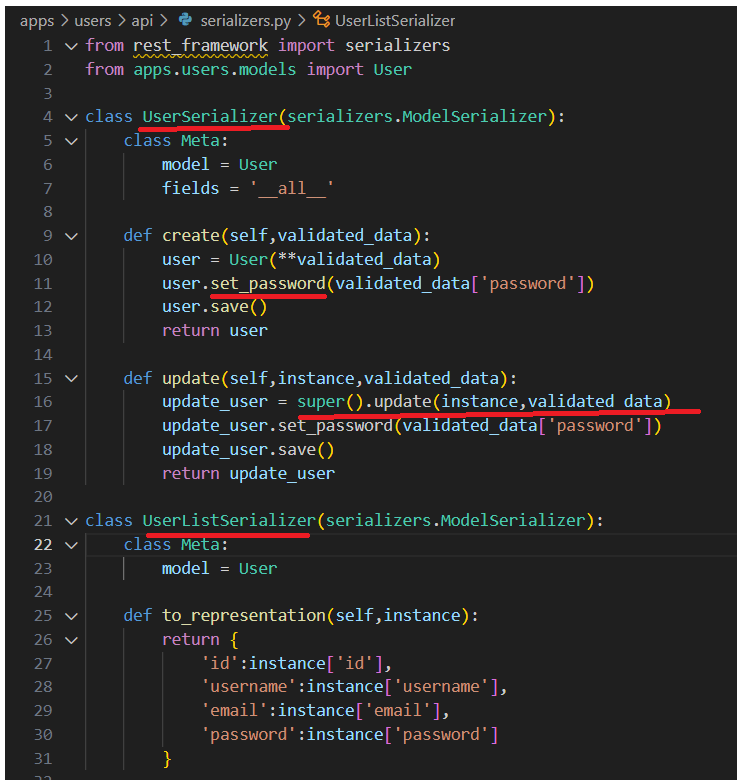




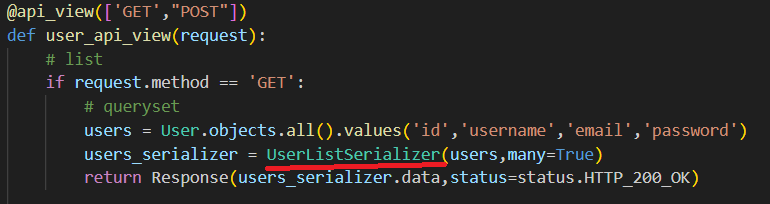
Pero esto me satura la memoria, porque me trae el **Queryset** con todos los campos y luego con **instance.key** represento los datos del queryset que necesito, pero no usaría todos los datos que consulte.

**Clase 14. ENCRIPTAR contraseña en un SERIALIZER**

Vamos a encriptar el **password** que el usuario asigna a cada instancia o registro del modelo ya sea cuando crea o actualiza. Pero primero se crea un **serializer** para listar y otro para crear y/o actualizar, así:

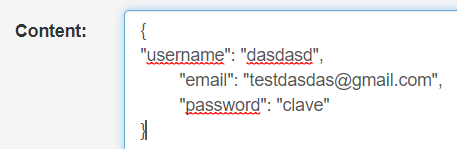


Y en la **api.py** el pequeño cambio



Entonces al crear una instancia nueva con **user.set\_password(validated\_data['password'])** encripto el campo del **password** con la función de Django **set\_password(),** y luego guarda la instancia.

Al crear



Se visualiza su **password** encriptada



Al actualizar una instancia primero actualiza el registro con todos sus campo con la línea **super().update(instance,validated\_data)** y luego encripta la contraseña con **update\_user.set\_password(validated\_data['password']) y** guarda, retornando la instancia.

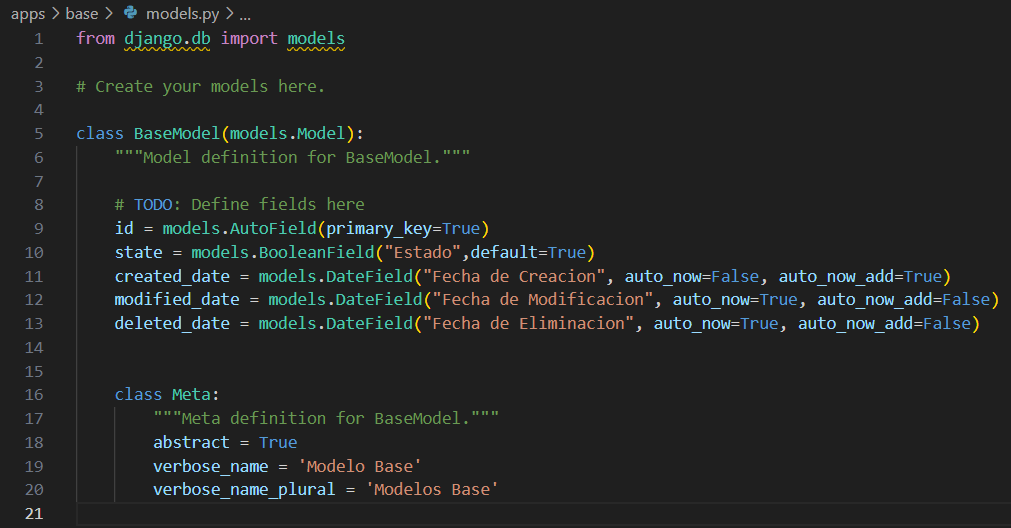
**Nota**: El método **set\_password()** cifra automáticamente la contraseña antes de almacenarla en la base de datos. Finalmente, se guarda el usuario en la base de datos con la contraseña cifrada.

**Clase 15. Aplicación Base y Productos**

En carpetas **apps** creamos la apps **products y base**

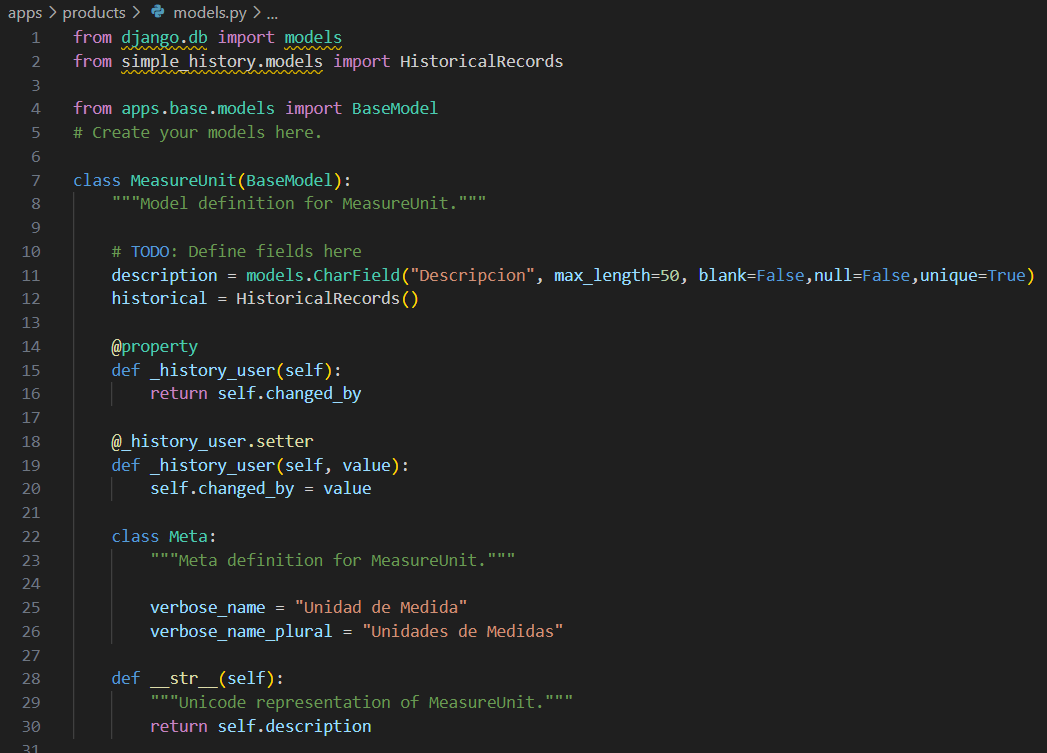
* **django-admin startapp products**
* **django-admin startapp base**

La app **base** sirve para crear bases que las otras apps usaran, algo así como el **core** del proyecto. En **base/models.py**

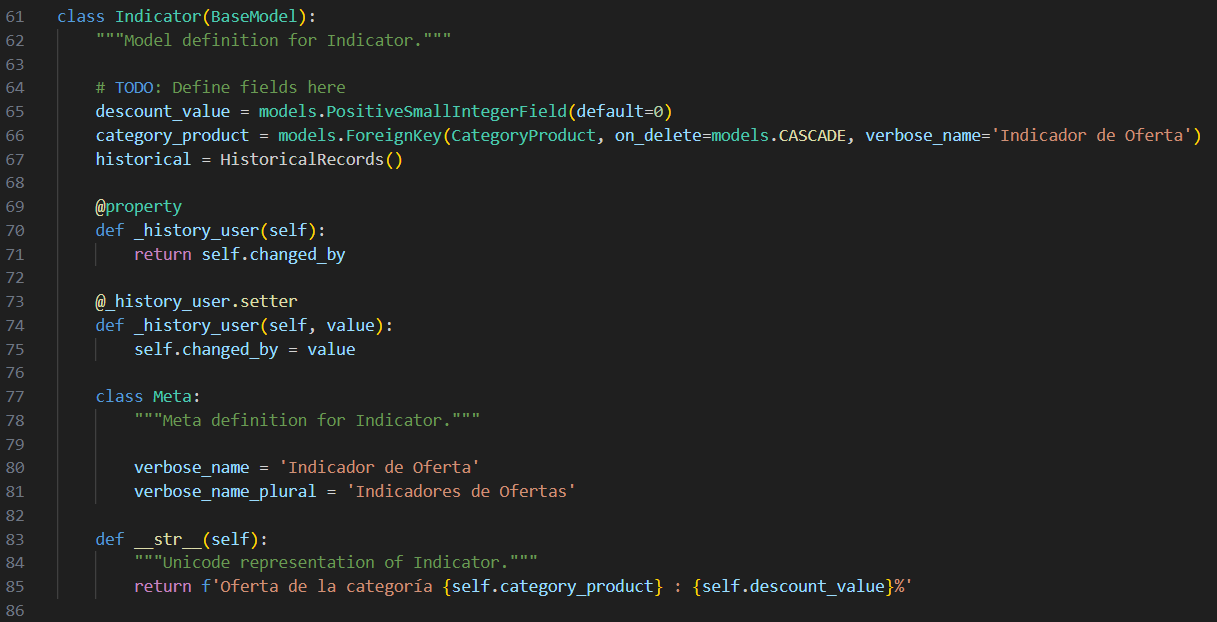


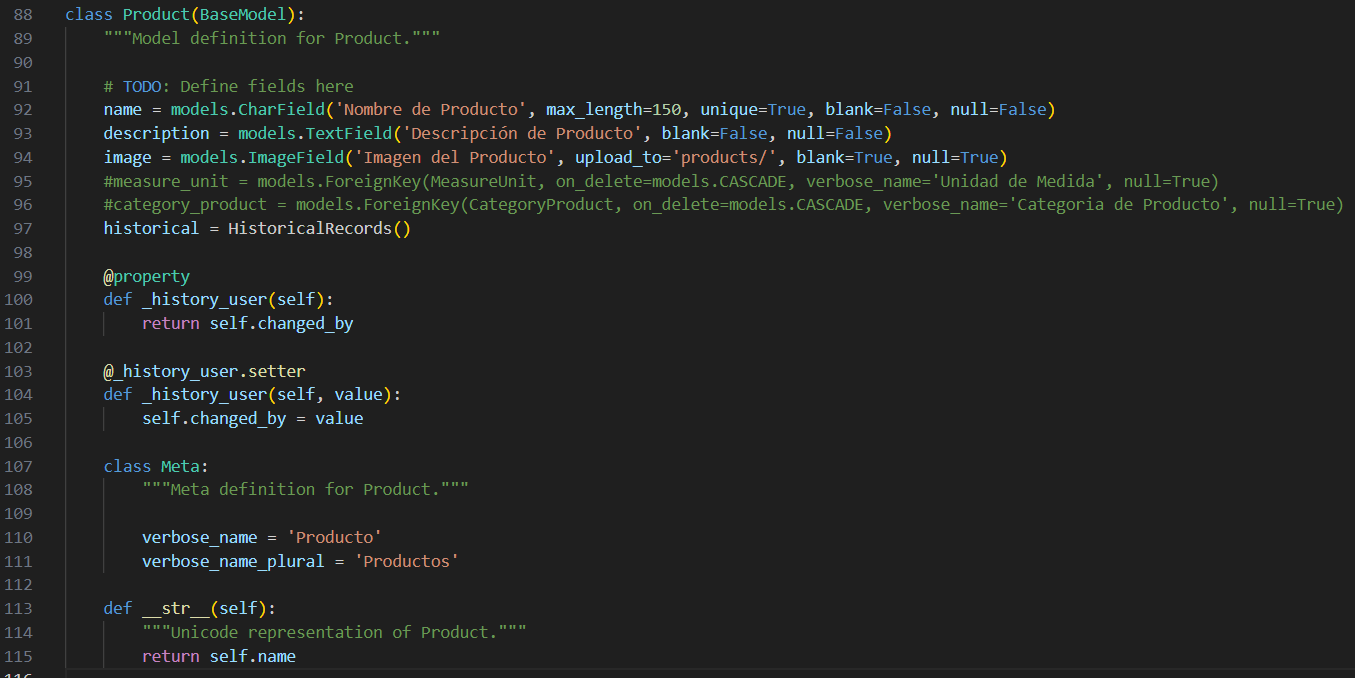
Donde el modelo base **BaseModel** llevará el registro del **estado, fecha de creación, fecha de modificación y fecha de eliminación** de todas las instancias o registros de los modelos que hereden de este.

Ahora la app **products** alberga la lógica de negocio del **ecommerce,** en **products/models.py** se crean 4 modelos, que heredan de **BaseModel**

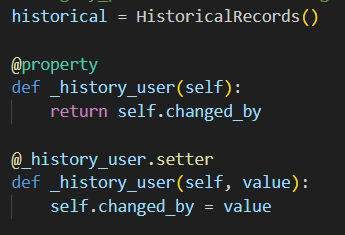
****



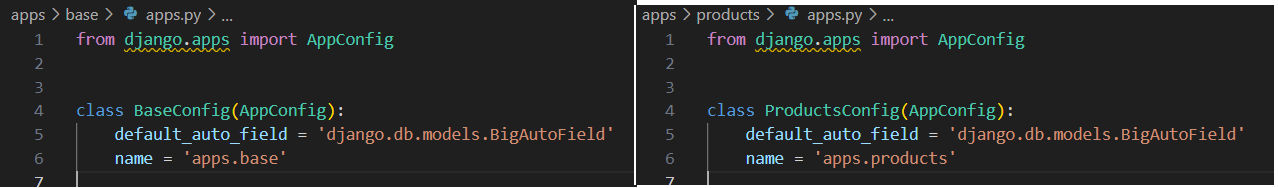




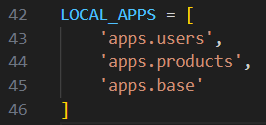
Los modelos llevan trazabilidad de los usuarios que gestionan los registros usando el modelos **HistoricalRecords** de la app de terceros **simple\_history**



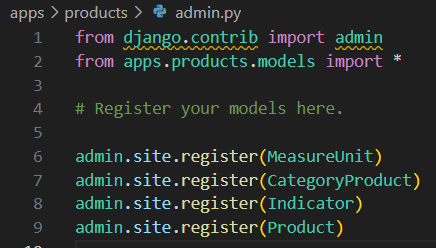
Se configurar los nombres de las apps en **apps.py**

****

y en **settings/base.py** se instalan las nuevas apps



Y se registran en el **admin.py**



Ahora se pueden hacer las migraciones

* **py manage.py makemigrations**
* **py manage.py migrate**

y ya en el Admin se observan los modelos



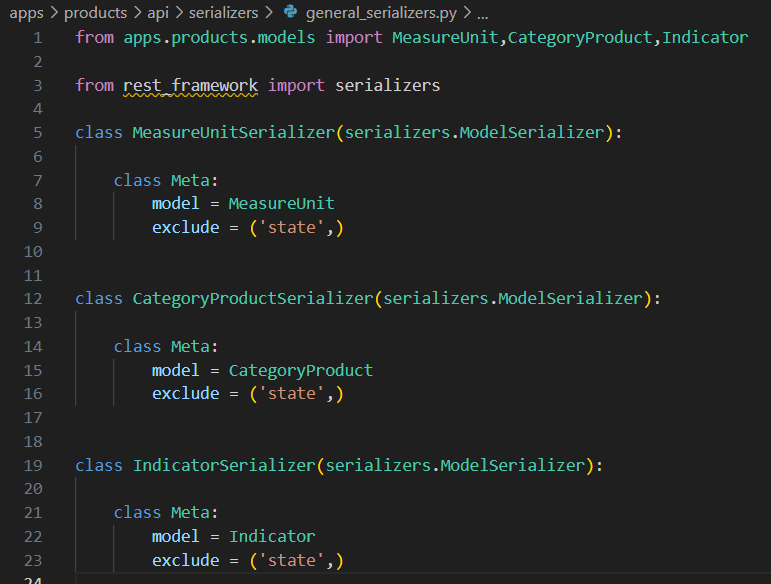
Para mayor información de los modelos de las apps que se plantearon se puede visitar el repositorio

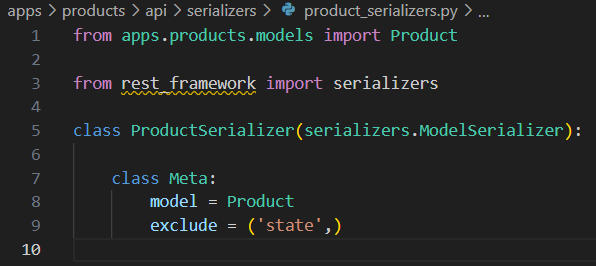
* <https://github.com/developerpe/ecommerce_rest/blob/master/apps/products/models.py>
* <https://github.com/developerpe/ecommerce_rest/blob/master/apps/base/models.py>

**Clase 16. Serializadores de Aplicación Productos**

De la carpeta **products/** eliminamos **views.py y test.py,** y creamos la carpeta **api.** Dentro de **products/api/** creamos las carpetas **serializers/ y views/.**

En  **products/api/serializers/** creamos los archivos **general\_serializers.py y product\_serializers.py,** y en ellos tenemos los siguientes serializadores para cada uno de los 4 modelos planteados en la app **products**





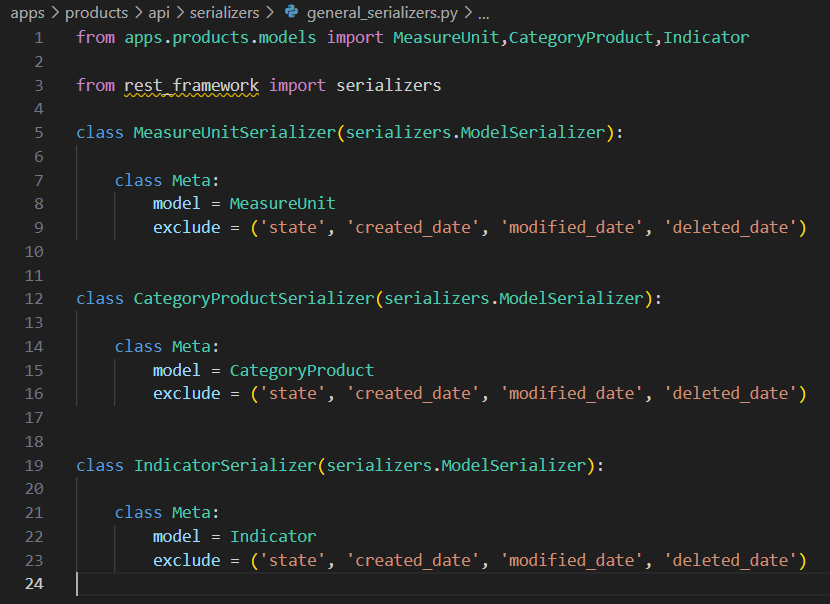
**Clase 17. Vistas Genéricas: LISTAPIVIEW**

Vamos a crear las **ListAPIView** de los 3 modelos usando **generics, generics** se refiere a un conjunto de clases y funciones predefinidas que proporcionan funcionalidad común para crear vistas y serializers en API basadas en REST. Las vistas genéricas son una forma conveniente de implementar patrones comunes en el desarrollo de API, como crear, leer, actualizar y eliminar (CRUD) operaciones. Al utilizar las clases genéricas, puedes escribir menos código repetitivo y aprovechar la funcionalidad integrada para realizar tareas comunes, como recuperar, crear, actualizar y eliminar registros de la base de datos.

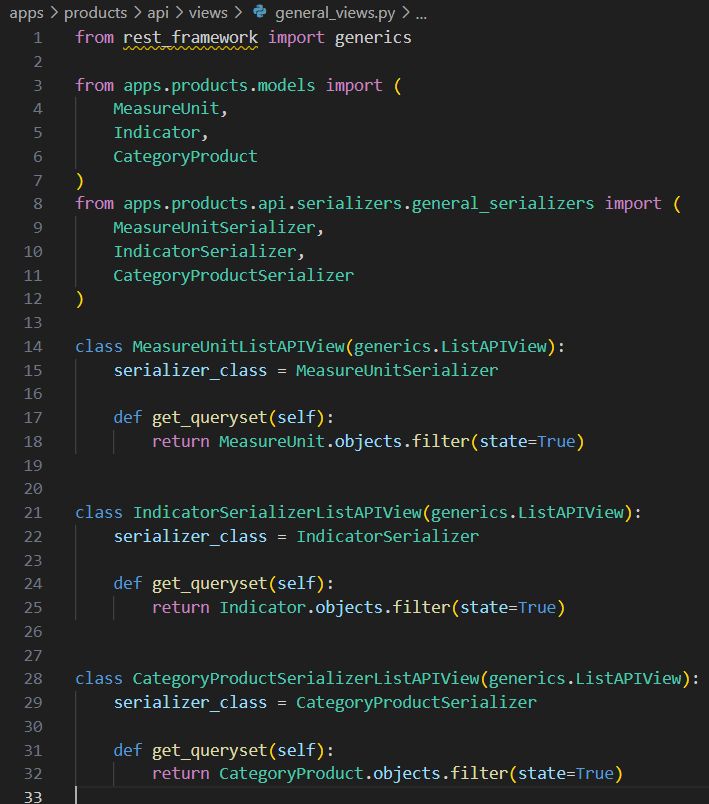
Algunas de las vistas genéricas más comunes en Django REST Framework son:

* **ListAPIView**: Proporciona una lista paginada de objetos.
* **RetrieveAPIView**: Recupera un objeto individual.
* **CreateAPIView**: Crea un nuevo objeto.
* **UpdateAPIView**: Actualiza un objeto existente.
* **DestroyAPIView**: Elimina un objeto existente.

Entonces vamos a mejorar primero los 3 **serializers**

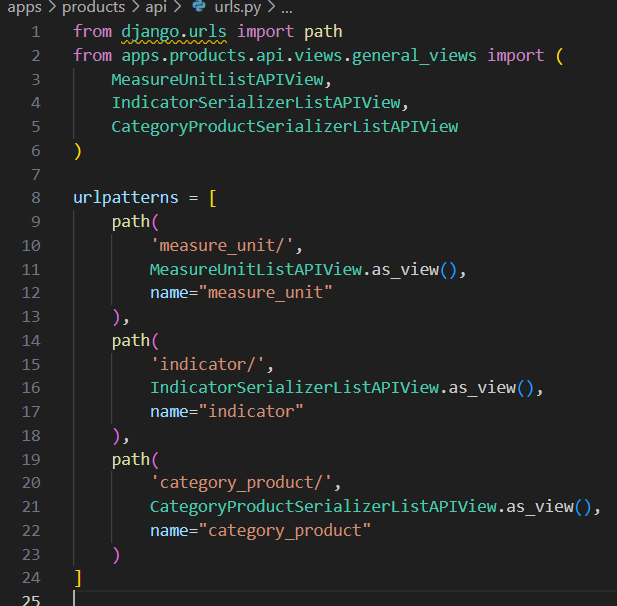
****

Ahora vamos a crear las vistas genéricas en **general\_views.py** dentro de la carpeta **products/api/views**

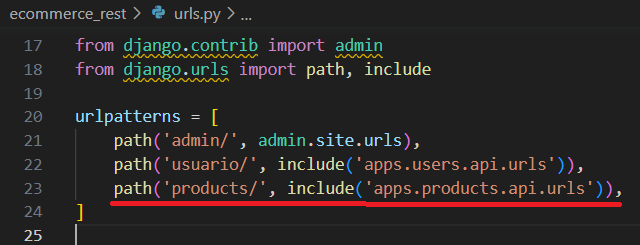
****

Note que esta vista basada en clase trabaja igual que las vistas basadas en funciones que vimos anteriormente, pero acá se entiende que la vista **ListAPIView** recibe como entrada el método **GET.**

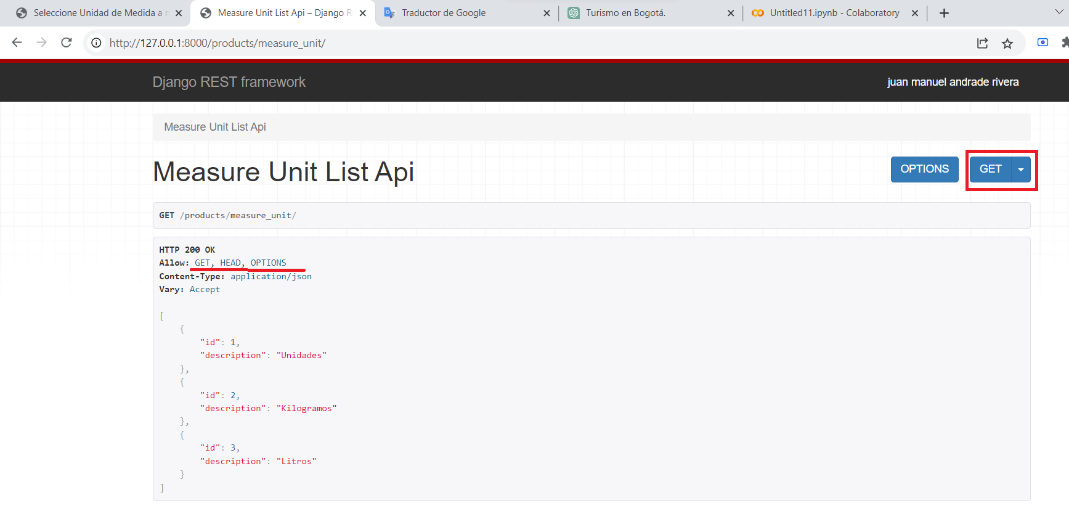
Ahora para las **urls** de estas vistas, se crea **urls.py** en la carpeta **products/api:**



Y estas urls se enlazan en **ecommerce\_rest/urls.py**



En Admin creamos 3 instancias del modelo **Unidad de medida,** en el navegador se ven reflejados



**Clase 18. LISTAPIVIEW General**

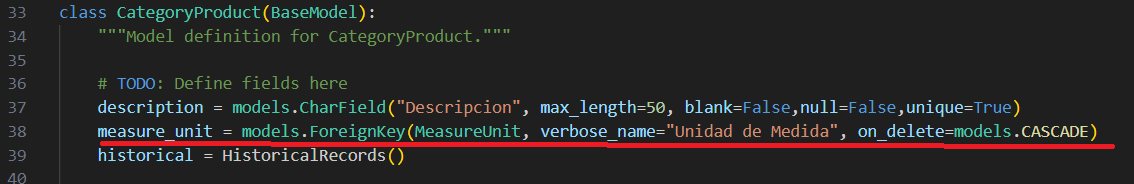
El autor recomienda la siguiente documentación muy completa sobre **DRF**

* <https://www.cdrf.co/>

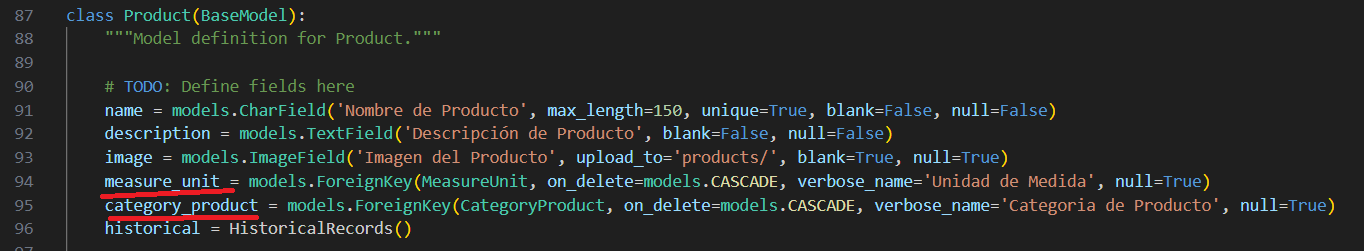
y sobre **DJANGO** teníamos

* <https://ccbv.co.uk/>

Vamos a corregir un error en el modelo, quitamos del modelo el campo en rojo

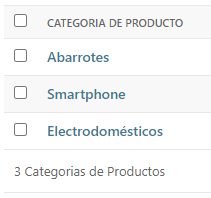


Y agregamos los 2 campos en el siguiente modelo



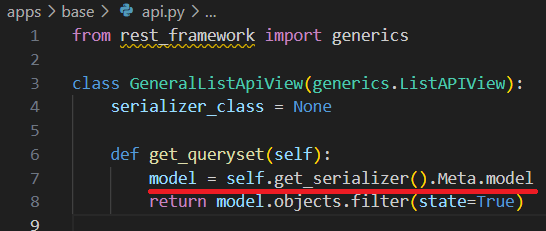
Y realizamos migraciones.

Creamos en el **Admin** las siguientes 3 categorías de productos

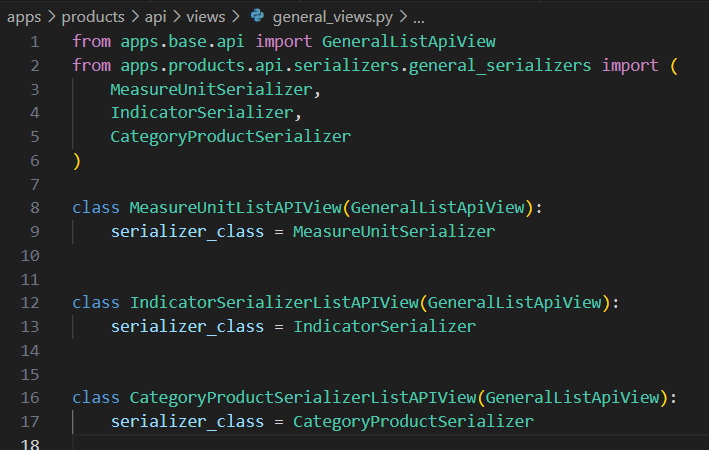


Por otro lado, se puede ver que en **general\_views.py** las 3 **listAPIView** creadas repiten su código, siendo idénticas las 3. Vamos a crear una **listAPIView** genérica personalizada en la app **base** para que hereden de ella.

En la app **base/** creamos **api.py** la siguiente



Y nuestras apis heredan de este



**Clase 19. Formas de Serializar Relaciones en Modelos**