Recibido

Ponme en django el modelo y la administración (admin.py) que cumpla con la siguiente entidad, el (\*) se refiere a que es obligatorio y el resto son opcionales, pero quisiera que se validara que la imagen fuese en jpg o png y con un tamaño menor a los 10mg

**Negocio**

* Nombre (\*)
* Imagen de Logo
* Slogan (una frase)
* Dirección/ubicación (\*)
* Descripción (\*)

Luciste

Si tengo un campo en un modelo que es ImageField como puedo usa el método clean del Modelo para validar que la imagen fuese en jpg o png y con un tamaño menor a los 10mg

<https://www.jobbank.gc.ca/reg/indregister1>

Estoy trabajando con django y tengo un modelo que se llama ImagenEspecial, el mismo esta compuesto por un campo imagen y un campo texto.

Tengo otro modelo llamado “Negocio” el cual puede de tener múltiples de estas ImagenEspecial, a su ves también cuento con otro modelo llamado ”Producto” que también puede tener múltiples de estas ImagenEspecial. El problema es que ImagenEspecial solo bebe de existir asociado a un de estos modelos, ”Negocio” o ”Producto”, como logro esto?, y la solución tienen algo que ver con el tipo models.ManyToOneRel?

De Python:

Cambian

Request.POST -> request.data (incluye archivo e imágenes enviados)

Request.GET -> query\_params

Dos variables dentro del request:

-accepted\_rendered

- accepted\_media\_type

From rest\_framework.response import Response

Siempre van a retornar un objeto Response

From rest\_framework import serializer

El serializador es una clase que representan a un modelo y en específico que campos son los que se van a serializar, esta extiende de “serializer.ModelSerializer”

Tiene una clase “Meta”

* model=ClaseModelo obvio
* exclude=(”campo1”, ”campo2”,…) tupla de campos que no se van a serializar
* fields=”\_\_all\_\_” o [ ”campo1”, ”campo2”,…] este o exclude, no ambos

Para las listas, se declara una clase que herede de “ViewSet”

* model=ClaseModelo obvio
* serielizerClass= clase serializador

luego un método para definir lo que pasa si se llama al get de este modelo y retornar una lista de las instancias disponibles ya serializadas

def list(self,request):

1ro se obtiene la lista usando al self.model y el request

lista=self.model.objects.all()

2do se serializa la lista usando a self.serielizerClass

listaSerializada=self.serielizerClass(lista, many=True) // de momento lo ultimo lo ignoro

3ro se retorna esta lista en una respuesta (Response)

Return Response(listaSerializada.data,status=200)

Como buena práctica se puede dividir al APPS INSTALLES en el BASE\_APPS

, LOCAL\_APPS, THIRD\_APPS

INSTALLED\_APPS = BASE\_APPS + LOCAL\_APPS + THIRD\_APPS

El archivo settings también se puede sustituir por una carpeta y en esta poner a base, local y production .py

En base esta la mayor parte de la configuración y luego en local y production esta la configuración de las carpetas, y las bases de datos:

DEBUG, ALLOWED\_HOSTS, DATABASES, STATIC, MEDIA

Si se va a serializar una sola instancia es

objetoSerializado=ClaseSerializadora(objetoSinSerializar)

Y si son varias se le agrega (…, many=True) como argumento a la llamada al constructor

listaSerializada=ClaseSerializadora(listaSinSerializar,many=True)

From rest\_framework.views import APIView

Para responder a un GET o POST se usa una clase que herede de APIView y defina los métodos

Get o post (self,request,parametros) y retorne un Response

from rest\_framework.decorators import api\_view

Se puede ligar a funciones apis ya creadas, para ello se les agrega el decorador @api\_view([‘POST’, o+ ‘’GET’ …]), aunque se defina dentro del decorador el o los tipos de método, hay que comprobarlo internamente con el request.Metod==’GET…’

Para hacer el efecto contrario, ósea se obtiene del request un json y este se serializa para luego obtener el objeto

1ro- Se le pasa el request.data al argumento data de constructor de la case serializadora

objetoSerializador=ClaseSerializadora(data= request.data)

2do- luego se hace una mínima validación dond ese comprueba que el json obtenido almenos cumple con la estructura deseada

esValido= objetoSerializador.is\_valid()

3ro- en caso de ser valido se manada a realizas la conversión de los datos con el método

objetoSerializador.save()

ese método también al mismo tiempo guarda la información en la base de datos

4to- se pueden ver los errores en formato json, estos se almacenan en caso de ocurrir dentro del atributo

erroresJSON= objetoSerializador.errors

Este puede ser enviado también en el data de un Response

Para el update se utiliza a PUT

1ro- instancia\_anterior=filtrar y obtener

2do- Variable\_Serializador=ClaseSerializador(instancia\_anterior,data=request.data)

Para el delete se utiliza DELETE

from rest\_framework import status

de hay se pueden obtener la mayoría de los códigos de status que se utilizan en el response

return Response(…,status=status.HTTP\_400\_BAD\_REQUEST)

Se pueden crear serializadores que no estén basados en modelos, y los campos disponibles para estos son los mismos que están disponibles para un modelo

Para la validación de los campos de un serializer, primero por cada campo se busca que exista un método:

def validate\_nombreCampo(self,value):

if validar:

raise serializers.ValidationError(“descripion”)

return value # si todo esta bien

y luego se llama si no hay errores a

def validate(self,data):

valorAEvaluar=data[‘’NombreAtributo”]

#lo validamos y disparamos el ValidationError

Return data #si todo esta bien

Para casos donde en se quiere pasar mas datos a la validación (antes de realizarse) o simplemente poder acceder a los a todos los datos existentes desde un método de validación, se le pasas al constructor del Serilizador otro argumento context= dic (puede ser el mismo data)

1ro- Variable\_serializador=ClaseSerilizador(data=data,context=data u otro dic)

2do- Luego

def validate\_nombreCampo(self,value):

dato=self.context[“dato”]

if validar:

raise serializers.ValidationError(“descripion”)

return value # si todo esta bien

Para sobreescribir la manera en que se guarda una instancia en la base de datos (solo al crear)

Def créate(self,validate\_data):

Instancia=ClaseModelo(\*\*validate\_data)

Return instancia

Para crear una instancia de un modelo rápido teniendo un dic con los valores de sus atributos

Instancia=ClaseModelo(\*\*dic\_conLosAtributos)

Para sobrescribir el método de salvar al editar (solo al editar)

Def update(self,instance,validate\_data):

Instance=super().update(instance,validate\_data) # si se quiere aprovecha rla edición automática de los campos

Instance.Atributo=validate\_data.get(“Atributo”, Instance.Atributo) # medoto del dic para que retorne un valor por defecto

Instance.save()

Return Instance

Para de un filtro a modelos obtener solo un objeto con solo ciertos campos

Instancia\_reducida=Clase\_Modelo.objects.filter(….).values(“Atributo1”,”Atributo2”)

Para manipular la forma en que se ve una instancia (por ejemplo para quitar la contraseña de la vista)

Def to\_represntation(self,instance):

Dato=instance[“atributo”]

Return diccionario

from rest\_framework import generics

Para las listas usar a vistas (clases) que hereden de generics. ListAPIView

Class VistaLista(generics. ListAPIView):

serializer\_class = ClaseSerializadora

def get\_queryset(self):  
 return self.serializer\_class.Meta.model.objects.all()

Cuando se quiere en ves de mostrar un atributo que sea llave como el id, sino mostrar toda la información de esta entidad, hay que declarar ese atributo en el serializador y darle como valor el serializador que representa a ese campo.

Class ClaseSerializador(serializer.ModelSerializer):

atributoLlave=ClaseSerializadoraAtributo()

Si lo que se quiere es mostrar su forma str por defecto (lo del atributo llave), se utiliza el tipo de campo se serializador serializer.StringRelatedField()

Class ClaseSerializador(serializer.ModelSerializer):

atributoLlave= serializer.StringRelatedField()

Para la vista de crear usar a vistas (clases) que hereden de generics. CreateAPIView

Class VistaCrear(generics.CreateAPIView):

serializer\_class = ClaseSerializadora

def get\_queryset(self):  
 return self.serializer\_class.Meta.model.objects.all()

Para la vista de ver una entidad usar a vistas (clases) que hereden de generics. RetrieveAPIView

Class VistaVer(generics. RetrieveAPIView):

serializer\_class = ClaseSerializadora

def get\_queryset(self):  
 return self.serializer\_class.Meta.model.objects.all()

Para la vista de eliminar una entidad usar a vistas (clases) que hereden de generics.DestroyAPIView

Class VistaEliminar(generics.DestroyAPIView):

serializer\_class = ClaseSerializadora

def get\_queryset(self):  
 return self.serializer\_class.Meta.model.objects.all()

Para la vista de editar una entidad usar a vistas (clases) que hereden de generics. UpdateAPIView

Class VistaEditar(generics. UpdateAPIView):

serializer\_class = ClaseSerializadora

def get\_queryset(self):  
 return self.serializer\_class.Meta.model.objects.all()

de manera adicional la vista editar admite el método PATCH (pasándole también el id) y lo que hace es que retorna la información (tipo get)

Para la vista de crear y listar usar a vistas (clases) que hereden de generics. ListCreateAPIView

Class VistaListarCrear(generics. ListCreateAPIView):

serializer\_class = ClaseSerializadora

def get\_queryset(self):  
 return self.serializer\_class.Meta.model.objects.all()

Para la vista de eliminar, editar y ver una entidad usar a vistas (clases) que hereden de generics. RetrieveUpdateDestroyAPIView

Class VistaEliminar(generics. RetrieveUpdateDestroyAPIView):

serializer\_class = ClaseSerializadora

def get\_queryset(self):  
 return self.serializer\_class.Meta.model.objects.all()

https://www.django-rest-framework.org/api-guide/viewsets/

Para unir el crud completo

class ClaseViewSet(viewsets.ViewSet):

def list(self, request):

pass

def create(self, request):

pass

def retrieve(self, request, pk=None):

pass

def update(self, request, pk=None):

pass

def partial\_update(self, request, pk=None):

pass

def destroy(self, request, pk=None):

pass

<https://www.django-rest-framework.org/api-guide/routers/>

Los ViewSet se registran (url) mediante routers

Esos se declaran en algún archivo aparte, seria buena costumbre llamarlo routers.py

from rest\_framework import routers

router = routers.SimpleRouter()

router.register(r'users', UserViewSet,basename=”suurl”)

router.register(r'accounts', AccountViewSet)

urlpatterns = router.urls

asdasdas

asdasd

asdasd

asdae

asdasdqweq