

¿Qué es un Servicio Rest?

Conociendo el potencial de Internet y de que se podía enviar información entre una y otra computadora, se creó una especie de arquitectura para este tipo de comunicación y esto era a través de lo que se conocía antes servicios web.

¿Qué son los servicios Web?

Pues el servidor base o el servidor donde estaba toda la información, o una computadora donde estaba toda la información del sistema. Podía crear servicios web para que los demás consuman estos servicios.

Sin embargo, a medida que fueron creciendo los sistemas web, bueno, esto era más sencillo porque ya no necesitabas un sistema de escritorio, sino simplemente construían un sistema web. Este ya estaba alojado en un servidor, tal cual los proyectos que se han hecho durante el curso

Solamente necesitaba un navegador para poder acceder al sistema y esta empieza a utilizar tu base de datos y ya no es necesario hacer esta comunicación entre XML y todo esto de aquí con un sistema web.

Pero ¿qué pasa si ahora necesito comunicar un sistema con otro independiente?

Mediante los servicios REST

Quiere decir que de aquí va a crear un servicio a los servicios web, solo que esta vez va a ser un servicio REST. Y la característica de este servicio REST. Es que es sencillo de implementar y ya no va a trabajar con formatos XML sino con archivos de formato JSON

¿Qué es JSON?

JSON es un formato de texto muy sencillo. Cuando me refiero a un formato de texto es, por ejemplo, si yo quería mostrar dentro de mi HTML. O bueno, si yo quería que en mi navegador se pinte algo, se muestre un esquema que hacíamos, escribíamos dentro de un archivo, un formato de esquemas, y esto lo hacíamos con HTML. Sabíamos cómo ubicar las cosas para que esto se muestre en el navegador.

Bien, pues un JSON es algo similar. Vamos a nosotros formar un texto, esta vez un texto algo así, con cabecera, con todo lo necesario cabecera, el cuerpo.

Pero es un texto que luego una aplicación, ya sea en JavaScript, ya sea una aplicación de Flutter, de Android, la que sea, va tomar todo ese texto de ahí y va a entender su formato, sabe que es un formato de tipo Jameson, va a entender y va a transformar esa información en objetos o clases que luego va a procesar para interactuar con nuestra base de datos.

Arquitectura REST - Servicios Web

Django REST nos pide en su arquitectura ciertos requisitos

- Tener bien separado lo que es el cliente del servidor

Es decir todo aquel proceso donde sea consulta de datos o guardar datos directamente interactuando con la BD tiene que ser un proceso exclusivo del servidor.

Sin embargo todo aquello donde se muestre una interfaz que interactúa con el usuario tiene que ser exclusivamente del cliente, lo que se conoce como frontend y el backend

- Peticiones o procesos sin estado es decir por ejemplo si se ha creado un servicio para que lo consuma el front ya sea de listar clientes, pues éste servicio no debe depender de ningún otro servicio o API

Característica Esencial ==>

- * Tiene una interfaz uniforme, es decir que todos van a acceder al servicio de la misma forma

- * Tiene un Sistema de Capas (desarrollo por capas) ==> hay que respetar las capas y el patrón de diseño

Métodos HTTP

GET- POST- PUT- DELETE

Códigos de Estado que responde REST

- 20X (correcto) ya sea 202, 200 , etc quiere decir que el proceso fue exitoso

- 30X (Caché, redirección) 301, 300, etc quiere decir que la info ya fue enviada varias veces y está en cache y no es necesario hacerlo de nuevo

- 40X (No se encontró) 404, 400 Que no ha encontrado nada/servicio

- 50X (error del servidor) 502, 501 Error del servidor, en caso de que por ejemplo en un listado no se puso correctamente el atributo de un modelo

Vista para Listar Personas

Se plantea el listar personas con un ListView y mostrarlo en un template html, lo cual ya se sabe cómo hacer y se lo hizo

Pero ¿qué sucede si ahora lo que me piden es un servicio REST que haga este mismo proceso de listar personas?

Éste servicio REST se va a mostrar bajo un formato JSON, entonces usualmente lo que hemos hecho es un proceso de llamada a la BD por el queryset, pasamos el contexto a un HTML, lo armamos y lo mostramos

Pero ahora necesitamos armar un archivo de tipo JSON para el consumo de otra aplicación a continuación lo haremos

Primer Servicio API - Listar Personas

Una vez instalado y añadido Django REST Framework en mi proyecto pues procedo a la vista donde quiero crear mi lista de tipo servicio, e importo la vista para la API

Creo mi clase, pero en este caso ya no especifico mi template porque se supone que ya no trabajamos con HTML, de igual manera no necesitaremos un `context_object_name` ya que el mismo es creado para llamarse en el HTML.

Entonces, vamos a crear un formato JSON por ende sí voy a necesitar un queryset de igual manera como se lo hacía anteriormente.

Sin embargo, el resultado del query necesitamos convertirlo en un archivo JSON y a este proceso se le llama **serialización del resultado**

También al proceso inverso, tomar un JSON y transformarlo en datos que se puedan entender

Django siempre exige que casi para todas sus clases pueda ser serializado, además debemos indicar en qué formato JSON se va a transformar por ende necesitamos un serializador donde le especificamos a Django que una lista viene de una forma o queremos que transforme una lista en un JSON

Para ello crearemos un `Serializers.py` que hace referencia a que dentro de este archivo vamos a tener todos los serializadores, los cuales se encargan de indicarle a Django en qué forma queremos que transforme los datos y que en qué forma los recupere

Dentro de éste serializer especifico el modelo al cual quiero serializar junto con sus atributos para posteriormente llamarlo a la vista

Aclaraciones sobre Json en DRF

Es el serializador el que nos muestra estructuradamente la información, casi en todas las vistas vamos a usar el serializador para recibir datos o transformar en JSON.

Ojo que de igual manera al hacer los serializadores con las vistas también se deben hacer las respectivas urls

Integrando un HTML con Funcionalidad JS en los templates

Se logra listar a los objetos de un modelo por medio de JavaScript haciendo una consulta al servidor para que este devuelva resultados a esa interacción... A esa comunicación se la conoce como Ajax

Para demostración lo que se hizo es usar Vue.js para poder listar las personas

Demostración de la interacción entre JS y Django REST

Se creó una nueva vista en la cual se manda de igual forma los parámetros por la URL y el archivo JS lo que hace es escuchar por así decirlo cada que se ingresa un caracter en el buscador y presentar los resultados, en pocas palabras una petición GET

CreateAPIView - Api para registrar persona

Cuando usabamos CreateView, a éste lo asociamos un formulario y este a su vez se relacionaba a un modelo, pero en éste caso no tenemos un formulario.

Entonces en éste caso lo que tenemos es serializadores, entonces lo usaremos ya que éste serializador se relaciona con el modelo, manipula un JSON.

¿Usamos un success_url?

NO!! porque éso de que es lo que va a pasar una vez se cree una nueva instancia le pertenece al frontend o a los que se encargan de construir la interacción con el usuario, en éste caso se está programando el backend

De igual manera se añade una url dentro de los urls.py

Al momento de ingresar la URL Django Rest nos presenta un formulario y con JS los valores los está volviendo en JSON y ése json se envía a la URL para ser guardado el nuevo objeto del formulario. Ojo que para todo ésto se está apoyando del serializador

DestroyAPIView - RetrieveAPIView

RetrieveAPIView es prácticamente lo mismo que el `DetailView`, es decir me sirve para ver el detalle de un objeto, obviamente de la misma forma en las url debo mandar el parametro del pk para especificar el objeto.

A diferencia del `DetailView`, aqui ya no se trabaja con el parametro `model` sino con el parametro `query set` donde dentro de éste parametro voy a especificar un conjunto de datos donde se va a hacer la búsqueda. (También se pueden hacer filtros)

DestroyAPIView de igual manera es similar al `DeleteView`, se mandan los parametros por la url, tiene un serializador la vista y un `queryset`

UpdateAPIView - Actualizar un registro

UpdateAPIView de igual manera es similar al `UpdateView`, se mandan los parametros por la url, tiene un serializador la vista y un `queryset`.

Sin embargo no es del todo bueno que para poder actualizar un objeto se tenga que estar mandando id por id , lo ideal sería mostrar los datos existentes para que posteriormente de entre todos ellos modificar alguno. Para ello utilizaremos el

RetrieveUpdateAPIView con el cual lograremos listar los datos del objeto para posteriormente modificarlos

Pero para los que no estén directamente conectados al modelo, cuál es el que representa aquí al `FormView`, usualmente en Django basado en clases se trabaja con el `FormView` y eso nos ayudaba a hacer el crud no necesariamente vinculado a un modelo, en pocas palabras se podía trabajar con varios modelos..

Entonces en Django REST claro que hay una representación para esto pero lo veremos más adelante

Serializers Simple

Similar a lo que se vió en modelos, así como existe el `model.Serializer`, también existe el `Serializer.Serializer` que no necesariamente está conectado a un modelo

Los tipos de datos que va a tener un `serializers.Serializer` es similar a los formularios

De igual manera se logró listar los registros de mi modelo `Persona`, la única diferencia es que ahora se está trabajando con una serialización no necesariamente vinculada a un modelo


Siempre en la vista para listar se va a tener un `query set`, pero el mismo debe estar representando directamente un modelo sino va a dar error

¿Pero qué sucede si quiero mandar otro parámetro en el serializador?

Pues simplemente lo pongo como atributo que el campo no es obligatorio, es decir le pongo un `default= False` ó un `required= False`

Agregando Modelos al Proyecto

¿Qué pasara si tengo la necesidad de agregarle más atributos pero a un `Serializer` pero que SÍ está conectado a un modelo? Pues simplemente lo añado antes del `Meta`

```
class PersonaSerializer2(serializers.ModelSerializer):  
    activo = serializers.BooleanField(default=False)   
    class Meta:  
        model = Person  
        fields = ('__all__')
```

```
[  
  {  
    "id": 2,  
    "activo": false,   
    "created": "2022-09-23T14:03:29.058616Z",  
    "modified": "2022-09-23T14:03:29.058616Z",  
    "full_name": "Juan Carlos",  
    "job": "Ingeniero",  
    "email": "juan.carlos@example.com",  
    "phone": "555-555-5555."  
  },  
  {  
    "id": 3,  
    "activo": true,  
    "created": "2022-09-23T14:03:29.058616Z",  
    "modified": "2022-09-23T14:03:29.058616Z",  
    "full_name": "Maria",  
    "job": "Diseñadora",  
    "email": "maria@example.com",  
    "phone": "555-555-5555."  
  }  
]
```

Esto es muy útil en el caso que tenga que añadir un valor que venga del frontend que necesite de ese atributo

Se modifico los models, se añadieron la clase `Hobby` y `Reunión`

Serializadores para campos ForeignKey y ManyToMany

Crearemos un serializador para nuestro modelo Reunion

```
"""Modelo para Reunion"""
persona= models.ForeignKey(Person, on_delete=models.CASCADE)
fecha = models.DateField()
hora = models.TimeField()
asunto = models.CharField(
    'Asunto de la Reunión',
    max_length=100
)
```

De igual forma se crea la vista, en la cual llamo al Serializador y también creo el respectivo URL

Reunion Api Lista

GET /api/reuniones/

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

```
[
  {
    "id": 2,
    "created": "2022-09-26T15:14:07.186115Z",
    "modified": "2022-09-26T15:14:07.186115Z",
    "fecha": "2022-09-26",
    "hora": "12:00:00",
    "asunto": "Materias del Ciclo",
    "persona": 5
  },
  {
    "id": 3,
    "created": "2022-09-26T15:28:28.608994Z",
    "modified": "2022-09-26T15:28:28.608994Z",
    "fecha": "2022-09-27",
    "hora": "18:00:00",
    "asunto": "GIIA-TA",
    "persona": 4
  }
]
```

Pero ¿qué pasaría si yo quiero que me liste todos los datos de mi FK persona?

Para poder mostrar todos los datos de un modelo en un serializador simplemente igualo el atributo a un serializador que ya existe exclusivamente para ese modelo y muestre todos los datos del modelo

```
class ReunionSerializer(serializers.ModelSerializer):

    persona = PersonSerializer() #Para poder mostrar todos los datos de
    # simplemente igualo el atributo a un serializador que ya existe ex

    class Meta:
        model = Reunion
        # fields = ('__all__')
        fields = (
            'id',
            'fecha',
            'hora',
            'asunto',
            'persona',
        )
```

```
[
  {
    "id": 2,
    "fecha": "2022-09-26",
    "hora": "12:00:00",
    "asunto": "Materias del Ciclo",
    "persona": {
      "id": 5,
      "created": "2022-09-23T14:04:36.633506Z",
      "modified": "2022-09-26T15:12:39.586753Z",
      "full_name": "Adrián Cabrera",
      "job": "Ingeniero",
      "email": "adrian.cabrera@unad.edu.co",
      "phone": "3100000000",
      "hobbie": [
        1,
        2,
        3,
        5
      ]
    }
  },
]
```

Sin embargo ésto es unicamente para cuando son relaciones con FK , en tanto que para las relaciones ManyToMany (hobbies con Persona) cambia únicamente ésto (many=True) en el serializador

```
class PersonaSerializer3(serializers.ModelSerializer):

    hobbies = HobbySerializer(many =True) #no olvidar los ()
    class Meta:
        model = Person
        fields = (
            'id',
            'full_name',
            'job',
            'email',
            'phone',
            'hobbies',
            'created',
        )
```

```
phone": "3100000000",
    "hobbies": [
      {
        "id": 1,
        "created": "2022-09-26T15:10:12.407081Z",
        "modified": "2022-09-26T15:10:12.407081Z",
        "hobby": "Escuchar Música"
      },
      {
        "id": 2,
        "created": "2022-09-26T15:11:31.647984Z",
        "modified": "2022-09-26T15:11:31.647984Z",
        "hobby": "Jugar Fútbol"
      },
      {
        "id": 3,
        "created": "2022-09-26T15:11:36.880834Z",
        "modified": "2022-09-26T15:11:36.880834Z",
        "hobby": "Programar"
      },
      {
        "id": 5,
        "created": "2022-09-26T15:11:41.748142Z",
        "modified": "2022-09-26T15:11:41.748142Z",
        "hobby": "Leer"
      }
    ],
    "created": "2022-09-23T14:04:36.633506Z"
  },
]
```

Se puede evidenciar que efectivamente listó todos los hobbies de la persona

Métodos en un Serializador - SerializerMethodField

Qué pasaría si además de añadir un atributo extra al serializador relacionado con el modelo pues quiero que éste atributo sea el resultado de operar los valores que sí tengo en base al modelo?

Pues mando un método, primero lo creo con el `serializers.SerializerMethodField()`, luego lo defino luego del meta anteponiendo la palabra `get`

```
class ReunionSerializer2(serializers.ModelSerializer):  
  
    #Qué pasaría si además de añadir un atributo extra al serializador relacionado con el modelo  
    #pues quiero que éste atributo sea el resultado de operar los valores que sí tengo en base al modelo  
    #Pues mando un método  
  
    fecha_hora = serializers.SerializerMethodField()  
    # persona = PersonSerializer()  
  
    class Meta:  
        model = Reunion  
        fields = (  
            'id',  
            'fecha',  
            'hora',  
            'asunto',  
            'persona',  
            'fecha_hora', #añado el método así no sea parte del modelo  
        )  
  
    def get_fecha_hora(self, obj): #se antepone el get al metodo  
        return str(obj.fecha) + ' - ' + str(obj.hora) #el obj hace referencia a la instancia
```

HyperlinkedModelSerializer

Pueden existir casos en los que al listar un objeto por medio de un FK se requiera no listar todos los atributos pero tampoco solo el id.

Cuando surja esta necesidad, generalmente se solicita cargar el modelo al que está relacionado, pero con un link o con una URL de referencia hacia cuando necesite cargar todos los datos

Para ello lo que se debe hacer es usar `LinkSerializer`, el cual al momento de hacer el serializador lo llamo así `serializers.HyperlinkedModelSerializer`

Añado un `extra_kwargs` el cual será un diccionario en el cual especificaremos qué atributo tipo modelo queremos que sea un link

```
'persona': {'view_name': 'persona_app:detalle-persona', 'lookup_field': 'pk'}
```

Especifico la vista que va a cargar

Especifico el parametro del objeto en la url

En caso de en mi `urls.py` tengo un slug se especificaría el mismo, si tengo id pues especifico id pero en este caso tengo pk por ende pongo pk

Paginación en los serializadores

Partimos de que nunca es bueno sobrecargar la información, entonces debemos ir paginando los registros obtenidos de una consulta

¿Cómo hago la paginación en un serializador?

En la vista donde vaya a hacer paginacion pues llamo a mi serializador y a parte a mi pagination_class que Tiene que ser igualado a un serializador o estructura que contemple paginacion

Luego, en mi serializers.py creo mi serializador de paginacion apoyandome de la librería pagination de DRF donde:

- Especifico el size
- Especifico el tamaño máximo de registros que quiero que me muestre

Finalmente creo mi url y efectivamente logra listar los registros paginandolos

Serializador para consultas especial de la ORM

¿Qué pasa si nos piden serializar resultados de la forma que quiero saber cuántas reuniones tengo o un reporte de reuniones con personas que tienen un específico trabajo?

Para ello voy a tener que hacer managers

RECORDAR

Si utilizamos el annotate, nos devuelve un conjunto o un query set agregado a este query set una nueva columna con el valor de la operación aritmética que nosotros estemos realizando. Sin embargo, si utilizamos el aggregate nos va a devolver un diccionario especificando únicamente el valor, sin otros datos más, únicamente el valor de la función aritmética que nosotros estemos indicando.

Sin embargo como es un diccionario le puedo agregar más elementos al diccionario producto de funciones/operaciones aritméticas

Hago normalmente mi manager en base al person_job y contando cada registro de reunion por el id.

Usamos el serializador ya antes creado de la Reunion, sin embargo ésto nos va a arrojar un error debido a que para que un serializador se realice correctamente debe estar exactamente de la forma como está cada registro o cada objeto.

Y no es igual el serializador al objeto recibido debido a que hemos creado un nuevo atributo con el annotate llamado cantidad al resultado, es decir, al query set, le estamos agregando un nuevo atributo, un nuevo atributo llamado cantidad. Además no olvidar que con el annotate obtenemos diccionarios o la estructura del objeto modificado

Entonces para solucionar ésto pues debo hacer es construir un serializador con la misma estructura que nos arroja el manager

```
class CountReunionSerializer(serializers.Serializer): # No
    persona__job = serializers.CharField()
    cantidad = serializers.IntegerField()
```

```
class ReunionManager(models.Manager):
```

```
# Si utilizamos el annotate, nos devuelve un conjunto o un
# una nueva columna con el valor de la operación aritmética
# Sin embargo, si utilizamos el aggregate nos va a devolver
# el valor, sin otros datos más, únicamente el valor de la
# estemos indicando. Sin embargo como es un diccionario l
# producto de funciones/operaciones aritméticas
```

```
def cantidad_reuniones_job(self):
```

```
    resultado = self.values()
```

```
    #Aquí voy a indicar en base a qué parametro/at
```

```
    #agrupar las reuniones y me haga la función a
```

```
    'persona__job' #agrupo en base al trabajo
```

```
    ).annotate(
```

```
        cantidad = Count('id') #id de la reunion
```

```
    )
```

```
    print('*****')
```

```
    print(resultado)
```

```
    return resultado
```

```
class ReunionByPersonJob(ListAPIView):
```

```
    """
```

```
        Lista Personas agrupadas por trabajo
```

```
    """
```

```
    serializer_class = CountReunionSerializer
```

```
    def get_queryset(self):
```

```
        return Reunion.objects.cantidad_reuniones_job()
```