API REST CON DJANGO REST FRAMEWORK

SERIALIZADORES Y VISTAS

El concepto de serializador en jango permite permite convertir objetos en tipos de datos comprensibles para javascript y marcos front-end. Dicho de otra forma, Django nos proporciona mecanismos para poder transformar los modelos (que al fin y al cabo son objetos) en otros formatos basados en texto como JSON o XML. Estos mecanismos son los llamados serializadores.

Veamos un ejemplo:

En el archivo views.py importamos el módulo

```
from django.core import serializers
```

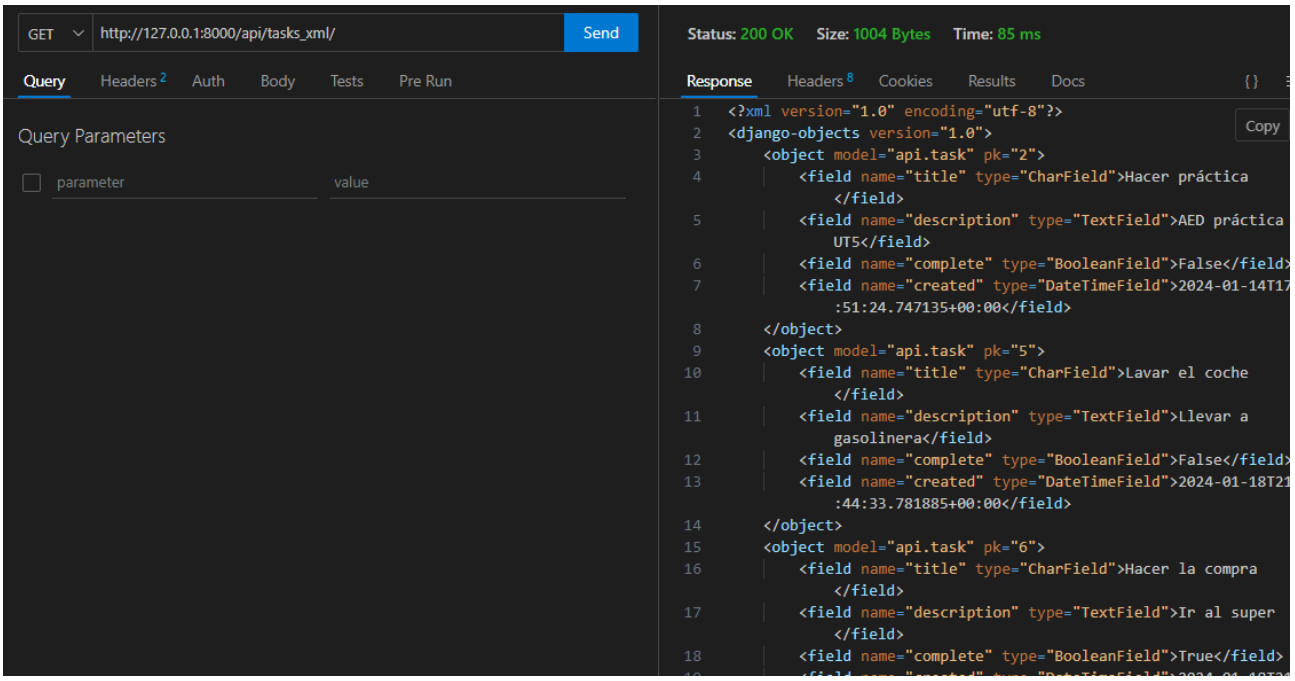
Generaremos una vista basada en clases que va a recibir todas las tareas y las convertirá en un XML.

```
class TaskViewXML (View):
    def get (self, request):
        data = serializers.serialize ("xml", Task.objects.all())
        return HttpResponse (data)
```

Utilizamos la función serializers.serialize para hacer la conversión a XML.
Necesitamos una nueva ruta para esta nueva clase. Por tanto el fichero api->> urls.py queda de la siguiente forma:

```
api > urls.py > ...
1   from django.urls import path, include
2   from .views import TaskView, TaskViewXML
3
4
5   urlpatterns = [path ('tasks/', TaskView.as_view(), name = 'task_list'),
6                  path ('tasks/<int:pk>', TaskView.as_view(), name = 'una_task'),
7                  path ('tasks/completadas/', TaskView.as_view(), name = 'completadas')
8                  path ('tasks_xml/', TaskViewXML.as_view(), name = 'tareas_xml'),
9   ]
```

Fíjate que se ha importado a nueva clase además de añadir la ruta.
Probamos la salida con Thunder Client



Fíjate en la ruta (la que habíamos puesto en el fichero api→urls.py) y fíjate que la salida es un xml bien formado que contiene todas las tareas de nuestra base de datos.

Con esta idea en mente de lo que es un serialización, lo que nos interesa es definir serializadores propios para nuestros modelos.

Para ello crearemos un nuevo archivo llamado **"serializers.py" dentro de la carpeta "api"** con el siguiente contenido:

```
api > serializers.py > ...
1   #Importamos la librería que nos permite trabajar con serializadores y nuestros modelos.
2
3   from rest_framework import serializers
4   from .models import Task
5
6   class TaskSerializer (serializers.ModelSerializer):
7       class Meta:
8           model = Task
9           fields = '__all__'
10          #campos = ('tittle', 'description', 'complete')
```

TaskSerializer está utilizando el ModelSerializer de Django Rest Framework, que genera automáticamente un serializador basado en el modelo Task. La clase Meta especifica el modelo a serializar (Task) e incluye todos los campos (fields = '__all__').

Si deseas incluir solo campos específicos en tu serializador, puedes utilizar el atributo fields y proporcionar una tupla con los nombres de los campos que deseas incluir como aparece en la línea comentada.

A continuación, trabajamos en el fichero **views.py**:

- Importamos del fichero serializers.py la nueva clase TaskSerializer
- Importamos de rest_framework el módulo viewset

```
from rest_framework import viewsets
from .serializers import TaskSerializer
```

El **módulo viewsets** en Django Rest Framework (DRF) proporciona una abstracción sobre las vistas de Django para facilitar el manejo de operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en una API. En lugar de definir las vistas manualmente, puedes utilizar viewsets para reducir la cantidad de código necesario y simplificar la implementación de la API.

Un viewset en DRF está diseñado para trabajar con modelos y proporciona métodos como list, create, retrieve, update y delete, que se correlacionan directamente con operaciones CRUD comunes. Estos métodos manejan las solicitudes HTTP GET, POST, PUT, PATCH y DELETE, respectivamente.

Y creamos la nueva clase

```
class TaskViewSet (viewsets.ModelViewSet):
    queryset = Task.objects.all()
    serializer_class = TaskSerializer
```

- TaskViewSet hereda de viewsets.ModelViewSet, que es una clase base proporcionada por DRF.
- queryset especifica la consulta que se utilizará para recuperar los objetos Task.
- serializer_class especifica el serializador a utilizar para convertir los objetos Task en datos que se pueden enviar como respuesta HTTP o recibir en una solicitud.

Al usar este TaskViewSet, DRF manejará automáticamente las operaciones CRUD para los objetos Task. Puedes configurar las rutas de URL correspondientes para este viewset en urls.py, y DRF se encargará de manejar las solicitudes y respuestas de manera consistente y eficiente.

Veamos cómo queda el fichero api→urls.py:

```python
api > urls.py > ...
from django.urls import include,  path
from rest_framework import routers
from .views import TaskView, TaskViewXML, TaskViewSet

router = routers.DefaultRouter()
router.register (r'tareas', TaskViewSet)


urlpatterns = [path ('tasks/', TaskView.as_view(), name = 'task_list'),
               path ('tasks/<int:pk>', TaskView.as_view(), name = 'una_task'),
               path ('tasks/completadas/', TaskView.as_view(), name = 'completadas'),
               path ('tasks_xml/', TaskViewXML.as_view(), name = 'tareas_xml'),
               path ('', include (router.urls)),
]
```

Un Router en DRF es una clase que te permite automáticamente generar las rutas de URL para las operaciones CRUD asociadas a un viewset. Esto simplifica la definición de las rutas en tus archivos urls.py y sigue las convenciones RESTful. Para poder usarlo es necesario importar el método “routers”.

La variable “**router”** es una instancia de routers.DefaultRouter(). Al usar este router, podemos registrar el TaskViewSet con el método register para generar automáticamente las rutas RESTful asociadas a las operaciones CRUD. En nuestro caso, hemos registrado el TaskViewSet bajo la ruta 'tareas'.

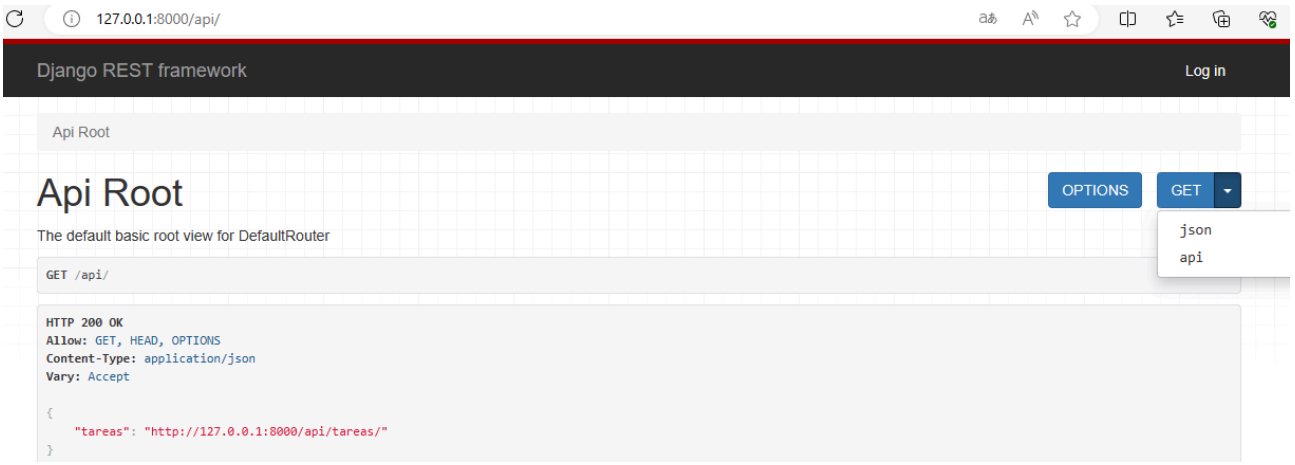
Cuando utilizas un router, puedes incluir las rutas generadas automáticamente en tus patrones de URL mediante la función include de Django. Esto facilita la organización y mantenimiento de tus rutas.

La finalidad de todo esto es demostrar cómo usando django rest_framework y las herramientas que ofrece bien configuradas, nos permite crear apis e implementar los métodos básicos sin apenas esfuerzo y sin tener que definirlos de forma manual. Vemos que sólo con poner en el navegador el nombre de nuestra api, podemos acceder a todas las rutas generadas sin más:

```python
path ('', include (router.urls)),
]
```

la ‘ ‘ hace referencia a nuestra api.

(Puedes probar todo ésto también desde el cliente)



Esto es lo que devuelve django en este momento sin tocar nada.

- Puedes acceder a todas las tareas haciendo click en el enlace en rojo.
- También te ofrece un método para insertar nuevas tareas