GUIA APIREST

Objetivo

Construir nuestro primer servidor de Django utilizando el patrón APIREST.

Enunciado

Configuración del entorno:

1. Activamos el env de django que hemos configurado en sesiones anteriores (**djansoServer**) de la siguiente manera:
    - Abre un terminal de **Comand Shell. No vale PowerShell de Windows.**
    - Localízate en esa carpeta:
      `cd djangoServer/Scripts`
    - Activa el entorno virtual:
      Actívate

2. A continuación, vamos a realizar la instalación de dos dependencias nuevas:
    - El framework djangoREST facilita la creación de aplicaciones REST.
      `pip install djangorestframework`
    - La extensión drf-spectacular permite documentar de forma automática los endpoints de las aplicaciones de django utilizando el estándar OpenAPI 3. Además, integra herramientas como Swagger para facilitar la interpretación de la documentación y poder interactuar con las aplicaciones.
      `pip install drf-spectacular`

3. El siguiente paso es crear el esqueleto de un nuevo proyecto. Usa el comando cd para ubicarte en la carpeta raíz que hayas elegido:
   `cd ../../`
   Y ejecutamos el comando:
   `django-admin startproject myFirstApiRest`

4. Abrimos el Visual Studio Code y abrimos la carpeta "myFirstApiServer" que se acaba de crear. Tendrá la siguiente estructura:

   myFirstApiRest /
   manage.py -> gestor de comandos de django
   myFirstApiRest /
   __init__.py
   settings.py -> configuración del proyecto
   urls.py -> definición de las urls del proyecto (referencia a las urls de las aplicaciones)
   asgi.py -> configuración de despliegue del servidor de django en entornos productivos
   wsgi.py -> configuración de despliegue del servidor de django en entornos productivos

5. Ahora crearemos la primera aplicación:
   `python manage.py startapp auctions`

6. El comando anterior crea las siguientes carpetas:

auctions /
__init__.py
admin.py -> permite registrar los modelos a los que se podrá acceder en la aplicación de admin que facilita django
apps.py -> define el arranque de la aplicación
migrations/ -> se almacenan las versiones de los cambios en los esquemas de los modelos
__init__.py
models.py -> representación en código de los esquemas de la base de datos
tests.py
views.py -> se define la lógica para cada tipo de petición

7. Ahora accedemos a myFirstApiRest/settings.py y buscamos la variable **INSTALLED_APPS**. Añadimos las dependencias de la nueva aplicación (auctions) y la de los frameworks que se han instalado en el paso 2. El resultado sería:

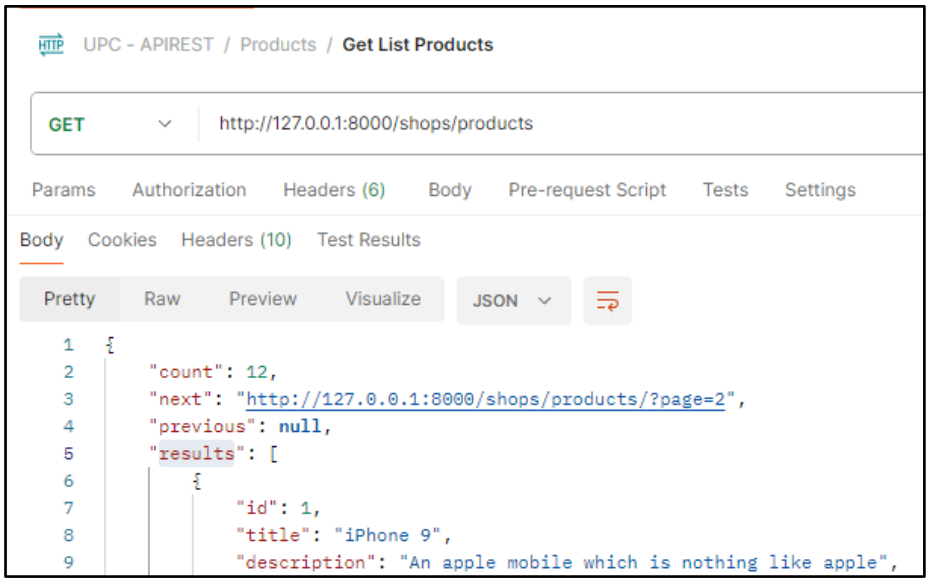
```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'auction', #para asociar la nueva aplicación (auction) al proyecto
    'rest_framework', #para importar el framework django REST al proyecto
    'drf_spectacular', #para importar la extensión drf spectacular al proyecto
]
```

8. Todavía en myFirstApiRest/settings.py, nos vamos al final del fichero y creamos una nueva variable: **REST_FRAMEWORK.** Esta variable es utilizada por el framework de REST para definir su configuración.
En este caso, vamos a delegar la lógica de paginación en el django para no tener que implementarla nosotros.
    - **DEFAULT_PAGINATION_CLASS***:* Se asocial a la clase responsable de paginación. Cogemos una proporcionada por el propio framework de REST.
    - **PAGE_SIZE:** Indica el número de elementos por página. Dado que en la base de datos tenemos 10 productos, lo establecemos a 5 subastas por página. Pero este valor queda a decisión del estudiante.

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 5,
}
```

Así, cuando solicitemos el listado de los productos será el propio framework de configurar el formato de la respuesta, indicando: el número total de productos (*count*), enlaces a las páginas anterior y siguiente (*next* y *previous*) y los productos de esa página (*results*). Aunque aún no

está configurado la aplicación para atender peticiones de este tipo, una vez finalizada los siguientes pasos, la paginación será parecida a esta:



9. Continuando en la variable **REST_FRAMEWORK** de myFirstApiRest/settings.py, vamos a indicar que el esquema que permita la generación de la documentación del api se haga por defecto. Para ello asociamos la variable **DEFAULT_SCHEMA_CLASS** con la que proporciona la extensión drf spectacular. El resultado final es:

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 5,
    'DEFAULT_SCHEMA_CLASS': 'drf_spectacular.openapi.AutoSchema',
}
```

10. El último cambio que vamos a realizar en myFirstApiRest/settings.py es añadir la variable **SPECTACULAR_SETTINGS,** que contiene la información básica para describir un api usando el estándar OpenAPI. Más información sobre las opciones de configuración aquí.

```
SPECTACULAR_SETTINGS = {
    'TITLE': 'API Auctions,
    'DESCRIPTION': 'Auctios web',
    'VERSION': '1.0.0',
    'SERVE_INCLUDE_SCHEMA': False,
}
```

11. Cambia la configuración de hora local a la de España: UTC+1 en invierno / UTC+2 en verano.

```
TIME_ZONE = 'Europe/Madrid'
```

Creando la versión APIREST

Una **APIREST** (Interfaz de Programación de Aplicaciones Representacional del Estado Transferido) es un conjunto de reglas y herramientas que permite a dos programas comunicarse entre sí a través de internet de manera eficiente y segura. Esta arquitectura se basa en los principios de REST, que utiliza operaciones HTTP estándar (GET, POST, PUT, DELETE) para realizar acciones sobre recursos, como datos o servicios, de manera uniforme y predecible. Sus principales características son:

- **Recursos**: En REST, todo es un recurso. En nuestra aplicación de tienda online, los recursos pueden ser cosas como usuarios y productos.

- **Identificadores únicos (URI):** Cada recurso tiene una identificación única, que se representa mediante una URL (Uniform Resource Locator). Por ejemplo:
  - /products para representar la colección de los productos que se venden en la tienda.

  El recurso se nombra en plural (products en lugar de product).
  No se deben incluir acciones en las URLs (por ejemplo, en lugar de /deleteProduct/{id}, se debería usar DELETE /products/{id}.
  Las URLs pueden reflejar la estructura jerárquica de los recursos. Por ejemplo:
  /users/{id}/posts para representar los posts de un usuario específico.

- **Métodos HTTP**: Utilizamos los métodos HTTP estándar para realizar operaciones en estos recursos:
  - **GET** para recuperar datos. Por ejemplo, **GET /products** para obtener todos los productos (lista).
  - **POST** para crear nuevos recursos. Por ejemplo, **POST /products** para agregar un producto nuevo.
  - **PUT** para actualizar recursos existentes. Por ejemplo, **PUT /products/{product_id}** para actualizar un producto específico. Es una actualización total del producto; es decir, hay que enviar todos los campos.
  - **PATCH** para actualizar algunos campos de recursos existentes. Por ejemplo, **PATCH /products/{product_id}**. Se mandan los campos que se quieren modificar.
  - **DELETE** para eliminar recursos. Por ejemplo, **DELETE /products/{product_id}** para eliminar un producto concreto.

- **Representaciones**: Los datos se intercambian entre el cliente y el servidor en forma de representaciones, como JSON o XML. Por ejemplo, cuando creamos una nueva subasta, podemos enviar los detalles de la subasta en formato JSON al servidor.

- **Sin estado:** REST es sin estado, lo que significa que cada solicitud contiene toda la información necesaria para procesarla. No se mantiene un estado de sesión en el servidor. Por lo tanto, cada solicitud es independiente y autónoma.

- **Interfaz uniforme**: REST promueve una interfaz uniforme entre el cliente y el servidor, lo que facilita la comprensión y el uso de la API. Esto significa que los clientes pueden interactuar con diferentes servicios de la misma manera, lo que simplifica el desarrollo y la integración.

"REST" se refiere a los principios arquitectónicos subyacentes, mientras que "RESTful" describe una implementación específica que sigue esos principios. Una API puede ser considerada RESTful si cumple con los requisitos y características definidos por REST.

Ahora que sabemos qué se conoce que es REST, prosigamos configurando la aplicación para que cumpla este diseño.

12. Accede a auctions/models.py para incluir la definición de los modelos. A diferencia de la sesión anterior, en esta ocasión tendremos dos modelos. Uno correspondiente a la subasta (*Auction*) y otro correspondiente a las categorías disponibles de una subasta (*Category*). Este tipo de relación corresponde a N:1, ya que una subasta podrá ser únicamente de una categoría, pero a cada categoría perteneces muchas subastas.

Por ejemplo, una subasta de un Iphone 12 tendrá como categoría “smartphones”. Pero a la categoría smartphones también pertenecen el Huawei Matebook X Pro y el Samsung Galaxy S10.

Para implementar esta relación, primero declararemos el modelo Category y el modelo Auctions. Posteriormente, añadiremos en Auctions el campo “category” como un ***models.ForeignKey().*** El primer parámetro indica el modelo al que hace referencia. El segundo corresponde con el campo que se crea en el modelo “Category” para hacer referencias a las subastas. El tercero, indica cómo se realizará el borrado de la relación si uno de las dos categorías desaparece.

Podéis encontrar más información sobre como implementar la relación N:1 en django en la documentación oficial.

```python
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=50, blank=False, unique=True)

    class Meta:
        ordering=('id',)

    def __str__(self):
        return self.name

class Auction(models.Model):
    title = models.CharField(max_length=150)
    description = models.TextField()
    price = models.DecimalField(max_digits=10, decimal_places=2)
    rating = models.DecimalField(max_digits=3, decimal_places=2)
    stock = models.IntegerField()
    brand = models.CharField(max_length=100)
    category = models.ForeignKey(Category, related_name='auctions',
on_delete=models.CASCADE)
    thumbnail = models.URLField()
    creation_date = models.DateTimeField(auto_now_add=True)
    closing_date = models.DateTimeField()

    class Meta:
        ordering=('id',)

    def __str__(self):
        return self.title
```

13. Tras definir el modelo, aplicaremos los comandos para que se creen los cambios en la base de datos:

- `py manage.py makemigrations auctions`
  - Este comando se encarga de detectar los cambios que ha habido en los modelos de la aplicación auctions. Si ha habido cambios, se genera un resumen de las operaciones (escritos en Python) que se harán en la base de datos (en nuestro caso el de sqlite) y se guardarán en auctions/migrations/.
  - **IMPORTANTE**: crea las operaciones, pero no las ejecuta. Eso lo hará con el siguiente comando (**migrate**).
  - El resultado de ejecutar el comando debe ser:

```
Migrations for 'auctions':
  auctions\migrations\0001_initial.py
    + Create model Category
    + Create model Auction
```

- `py manage.py migrate`
    - Este comando es el encargado de aplicar todas las migraciones de todos los aplicativos del proyecto.
    - En esta ocasión, al ser la primera vez que ejecutamos esta operación (*migrate*) se aplicarán no solo las migraciones de la aplicación de "***auctions***" sino también las de ***admin, auth, contenttypes, sessions***. Y se creará la "db.sqlite3".

14. Para comprobar que se ha realizado correctamente las operaciones, podemos acceder a la base de datos. En este caso, accedemos por el terminal:

    - `py manage.py shell`

    De esta forma accedemos a una consola de Python que permite hacer consultas sobre el modelo. Para realizar la inserción de un conjunto de productos que usar de ejemplo en la base de datos podéis utilizar el script proporcionado con el guión: "script bd apirest.txt".

    IMPORTATE incluir la importación del modelo: "from auctions.models import Category, Auction".

Ejercicio 1.

Una vez ejecutado todas las queries en la Shell de Django, responde:

- ¿Qué campo del modelo no estamos informado en cada subasta de ejemplo? ¿Por qué?
- ¿Qué valor se le ha asignado? Puedes usar la query 5 del script para obtener los valores. Y usar la función strftime para facilitar la lectura.

Ejercicio 2.

Si te fijas en query 7, se está llamando al campo "auctions" del modelo Category. Pero ese campo no se ha dado de alta al crear las tres categorías de ejemplos ni está definido en el modelo Category en código.

- ¿Dónde se define ese campo y cómo? ¿Qué representa?

15. Accede a myFirstApiRest/urls.py para añadir la referencia a las urls de la aplicación de **auctions**.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/auctions/', include('auctions.urls')),
]
```

16. A continuación, crea el fichero auctions/urls.py. Aquí se definirán las urls (también llamado endpoints) siguiendo el formato REST que hemos visto anteriormente:
    - categories/:
        - GET: consulta de todas las categorías disponibles (lista).
        - POST: crear una nueva categoría.
    - categories/<int:pk>:
        - GET: obtiene una categoría específica por id.
        - PUT / PATCH: actualiza (total o parcialmente) una categoría ya existente.
        - DELETE: borra la categoría específica.
    - auctions/:
        - GET: consulta de todas las subastas disponibles (lista).
        - POST: crear una nueva subasta.
    - auctions/<int:pk>:
        - GET: obtiene una subasta específica por id.
        - PUT / PATCH: actualiza (total o parcialmente) una subasta ya existente.
        - DELETE: borra una subasta específica.

    Estas urls son la base de la funcionalidad que proporcionará nuestra aplicación **auctions** y es el contrato acordado entre cliente y servidor.

    Llevado a código, a cada url se le asocia una vista (contiene la lógica de lo que se espera que haga y las consultas a base de datos) y un nombre. El resultado será:

```
from django.urls import path
from .views import CategoryListCreate, CategoryRetrieveUpdateDestroy, Auction-
ListCreate, AuctionRetrieveUpdateDestroy

app_name="auctions"
urlpatterns = [
    path('categories/', CategoryListCreate.as_view(), name='category-list-create'),
    path('categories/<int:pk>/', CategoryRetrieveUpdateDestroy.as_view(), name='category-detail'),
    path('', AuctionListCreate.as_view(), name='auction-list-create'),
    path('<int:pk>/', AuctionRetrieveUpdateDestroy.as_view(), name='auction-detail'),
]
```

Serializadores

En Django, los **serializers** son componentes que permiten convertir los modelos de la base de datos y otros tipos de datos de Python en formatos de intercambio de datos como JSON, XML o incluso HTML, y viceversa. Son esenciales en el desarrollo de APIs (Interfaces de Programación de Aplicaciones) en Django para facilitar la comunicación entre el frontend y el backend, o entre diferentes sistemas.

Aquí hay algunas características clave de los serializers de Django:

- **Serialización de datos:** Los ***serializers*** toman objetos complejos de Python, como instancias de modelos de Django, y los convierten en formatos de datos simples y estructurados como JSON o XML. Esto facilita el intercambio de datos entre la aplicación Django y otros sistemas, como aplicaciones frontend o servicios web.

- **Deserialización de datos:** Los ***serializers*** también pueden realizar el proceso inverso, es decir, tomar datos en formato JSON o XML y convertirlos en objetos de Python, como instancias de modelos de Django. Esto es útil cuando se reciben datos de una solicitud HTTP y se necesitan convertir en objetos para su procesamiento en el backend.
- **Validación de datos:** Los ***serializers*** pueden realizar la validación de datos durante el proceso de deserialización, asegurando que los datos recibidos cumplan con ciertos criterios antes de ser procesados por la aplicación. Esto ayuda a mantener la integridad de los datos y a prevenir errores en la aplicación.
- **Personalización**: Los ***serializers*** de Django permiten una gran flexibilidad y personalización. Los desarrolladores pueden definir campos específicos a ser incluidos o excluidos en la serialización, aplicar transformaciones a los datos antes de ser serializados o deserializados, y definir reglas de validación personalizadas.

17. Creamos un fichero nuevo llamado **serializers.py en auctions/** y definiremos un serializador por cada una de las operaciones (endpoints) de nuestro aplicativo definifos en las urls.

    Hay dos tipos de serializadores: manual o de modelo. En nuestro caso, usaremos el de modelo (**ModelSerializer**) ya que categoría y subasta están modeladas. A cada serializador le indicaremos la clase que tiene que transformar (mediante la propiedad **model**) y los campos que debe mostrar (**fields**).
    Es importante indicar que en este apartado podemos crear, modificar o eliminar los campos que mostramos al cliente, mediante la configuración de "fields".

```
from rest_framework import serializers
from .models import Category, Auction

class CategoryListCreateSerializer(serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = ['id','name']

class CategoryDetailSerializer(serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = '__all__'

class AuctionListCreateSerializer(serializers.ModelSerializer):
    class Meta:
        model = Auction
        fields = '__all__'

class AuctionDetailSerializer(serializers.ModelSerializer):
    class Meta:
        model = Auction
        fields = '__all__'
```

Vistas

En django existen tres tipos de vistas. **En nuestro proyecto usaremos las ViewSets,** que manejan solicitudes HTTP y devuelven respuestas utilizando métodos específicos de la clase. Estas clases proporcionan una forma estructurada y reutilizable de definir la lógica de las API REST.

Existen una vista para cada una de las operaciones CRUD (**CreateAPIView, RetrieveAPIView, UpdateAPIView, DestroyAPIView**) y también combinaciones de ellas. Para adaptarnos a REST (y por ende, a las urls y los serializadores definidos con anterioridad), definiremos las siguientes vistas:

- **ListCreateAPIView**:
  - Maneja las operaciones de listar (GET) y crear (POST) recursos.
  - Proporciona una lista paginada de recursos y permite la creación de nuevos recursos.
  - Útil para endpoints que necesitan mostrar una lista de recursos y permitir la creación de nuevos recursos.

- **RetrieveUpdateDestroyAPIView**:
  - Maneja las operaciones de recuperar (GET), actualizar (PUT) y eliminar (DELETE) un recurso individual.
  - Combina las funcionalidades de las vistas RetrieveAPIView, UpdateAPIView y DestroyAPIView.
  - Útil para endpoints que necesitan mostrar un recurso específico y permitir su actualización y eliminación.

18. Traducimos el concepto explicado de vistas a código en el fichero *auctions/views.py:*

```
from rest_framework import generics
from .models import Category, Auction
from .serializers import CategoryListCreateSerializer, CategoryDetailSerial-
izer, AuctionListCreateSerializer, AuctionDetailSerializer

class CategoryListCreate(generics.ListCreateAPIView):
    queryset = Category.objects.all()
    serializer_class = CategoryListCreateSerializer

class CategoryRetrieveUpdateDestroy(generics.RetrieveUpdateDestroyAPIView):
    queryset = Category.objects.all()
    serializer_class = CategoryDetailSerializer

class AuctionListCreate(generics.ListCreateAPIView):
    queryset = Auction.objects.all()
    serializer_class = AuctionListCreateSerializer

class AuctionRetrieveUpdateDestroy(generics.RetrieveUpdateDestroyAPIView):
    queryset = Auction.objects.all()
    serializer_class = AuctionDetailSerializer
```

Donde ***queryset*** contiene los resultados de la consulta a base de datos y ***serializer_class*** referencia al serializador asociado.

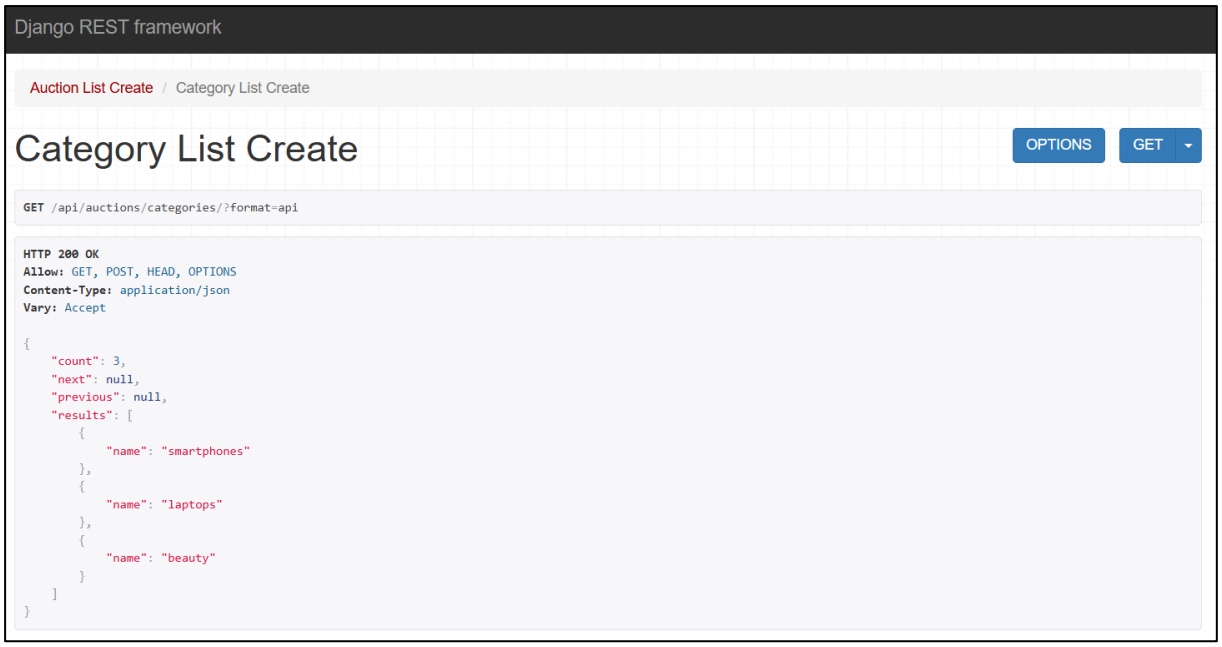
En conclusión, son las vistas (capa de lógica) las intermediarias entre realizar las actualizaciones en base de datos (capa de datos) y lo que el cliente recibe mediante la asociación del serializador (capa de presentación).

19. Ahora que tenemos implementada todas las capas de la APIREST de la tienda, podemos probar el resultado.
    - **`py manage.py runserver`**
    - Accede a http://127.0.0.1:8000/api/auctions/categories/

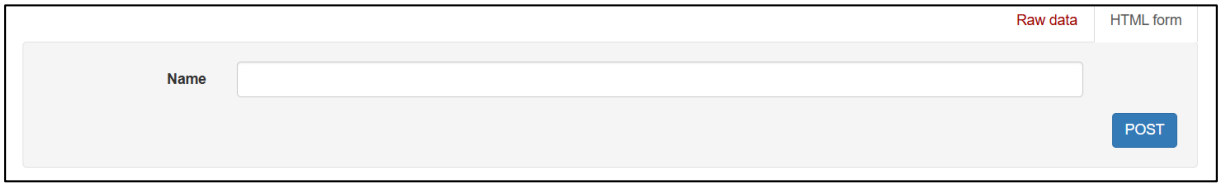
    Se esperaría obtener un json en formato texto plano:


```
{"count":3,"next":null,"previous":null,"results":[{"name":"smartphones"},{"name":"laptops"},{"name":"beauty"}]}
```


Pero el resultado es mucho mejor, ya que el framework de django REST proporciona una interfaz visual mucho más amigable:



Adicionalmente, al final de la página actual web actual, podemos encontrar un formulario que nos permite crear una nueva categoría (POST de la vista de ***`CategoryListCreate`***).



Ejercicio 3.

Prueba a crear una categoría nueva llamada “furniture”. ¿A parece la nueva categoría de muebles si vuelves a consultar el listado de categorías?

20. Podemos consultar los otros servicios que expone nuestra APIREST:
    - Accede a http://127.0.0.1:8000/api/auctions/5/

Ejercicio 4.

Vamos a realizar una modificación de la subasta. Primero, cambia el precio de la subasta de 1399 a 1400 euros. A continuación, cambia la fecha de cierre (*closing_date*) a hoy. Finalmente, haga clic en "PUT" para confirmar la modificación de la subasta.

- ¿Por qué se muestra el campo de fecha de creación (*creation_date)* pero no se ofrece en el formulario de edición?
- ¿Hay diferencia entre el formato de la fecha de creación y la del cierre?
- Modifica el fichero de auctions/serializers.py para incluir las líneas en negrita. Una vez hecho, vuelva a acceder a http://127.0.0.1:8000/api/auctions/5/ y observe cómo han cambiado estos campos.

```
class AuctionListCreateSerializer(serializers.ModelSerializer):
    creation_date = serializers.DateTimeField(format="%Y-%m-%dT%H:%M:%SZ",
read_only=True)
    closing_date = serializers.DateTimeField(format="%Y-%m-%dT%H:%M:%SZ")

    class Meta:
        model = Auction
        fields = '__all__'

class AuctionDetailSerializer(serializers.ModelSerializer):
    creation_date = serializers.DateTimeField(format="%Y-%m-%dT%H:%M:%SZ",
read_only=True)
    closing_date = serializers.DateTimeField(format="%Y-%m-%dT%H:%M:%SZ")

    class Meta:
        model = Auction
        fields = '__all__'
```

Ejercicio 5.

El estado de una subasta indica si se puede seguir pujando o no. Este estado se puede representar con un booleano y se calcula en función de si la fecha de cierre (*closing*_date) es anterior o posterior a la fecha actual. Como es un campo que se calcula dinámicamente (en función de la fecha y hora actuales) no debe ser almacenado en base de datos si no que debe generar en el serializador. Por ello, incluye en el serializador las siguientes líneas en negrita:

```
from rest_framework import serializers
from django.utils import timezone
from .models import Category, Auction

#Los serializadores de Category.

class AuctionListCreateSerializer(serializers.ModelSerializer):
    creation_date = serializers.DateTimeField(format="%Y-%m-
%dT%H:%M:%SZ", read_only=True)
    closing_date = serializers.DateTimeField(format="%Y-%m-%dT%H:%M:%SZ")
    isOpen = serializers.SerializerMethodField(read_only=True)

    class Meta:
        model = Auction
        fields = '__all__'

    def get_isOpen(self, obj):
        return obj.closing_date > timezone.now()

class AuctionDetailSerializer(serializers.ModelSerializer):
    creation_date = serializers.DateTimeField(format="%Y-%m-
%dT%H:%M:%SZ", read_only=True)
    closing_date = serializers.DateTimeField(format="%Y-%m-%dT%H:%M:%SZ")
    isOpen = serializers.SerializerMethodField(read_only=True)

    class Meta:
        model = Auction
        fields = '__all__'

    def get_isOpen(self, obj):
        return obj.closing_date > timezone.now()
```

- Con el ***SeriealizerMethodField*** indicamos que se trata de un campo que no se incluye en el modelo. ¿Por qué usamos la opción "read_only=True"?

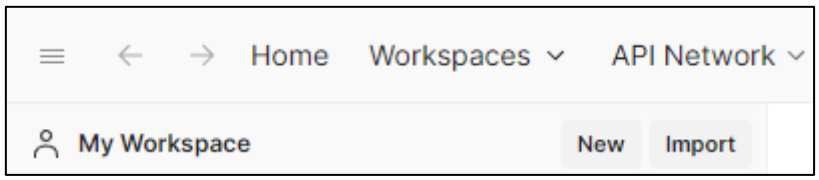
Ejercicio 6

Accede a http://127.0.0.1:8000/api/auctions/ y crea una nueva subasta. Los datos de la subasta quedan a elección del estudiante; el único requisito es que la fecha de cierre se establezca a dos minutos desde el momento de la creación. Es decir, si se va a crear la subasta el día 24/03/2025 a las 17:00, la fecha de cierre debe ser el día 24/03/2025 a las 17:02.

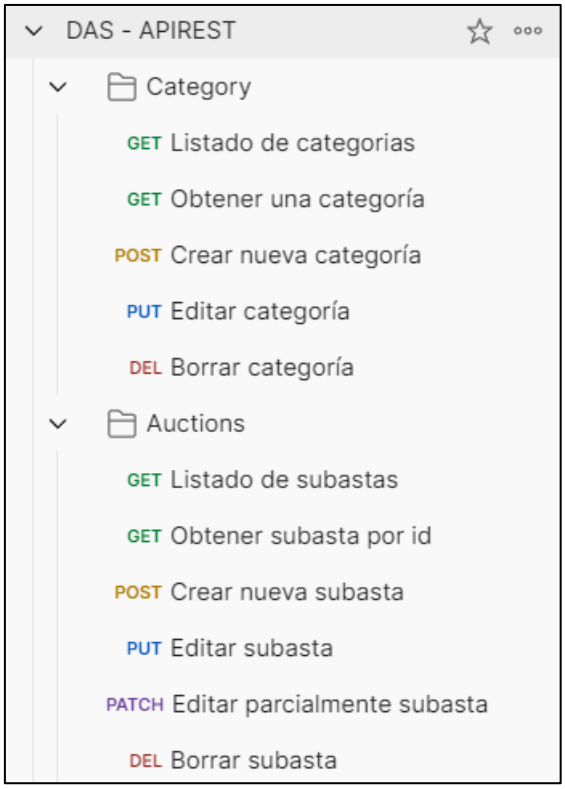
- Observe que la categoría se muestra como un desplegable en la creación.
- Espere 2 minutos y observe si el estado del cierre (*isOpen*) cambia de true a false.
- Borra la subasta una vez realizado el paso anterior.

21. Se proporciona una colección de Postman para poder ver otra forma de interactuar con el servidor: DAS - APIREST.postman_collection.json.

    Una vez iniciado sesión en la aplicación de Postman, añadir la colección adjunta haciendo clic en el botón importar:



    El resultado debería ser el siguiente:



    Se incluye un ejemplo de cada una de las operaciones CRUD que hemos definido para nuestros recursos (categories y auctions).

OpenAPI

Es una especificación para describir, producir, consumir y visualizar APIs RESTful. Proporciona una manera estándar y de fácil comprensión para que desarrolladores y equipos de desarrollo de software describan la funcionalidad de sus API web.

- **Descripción de la API**: OpenAPI permite describir los endpoints de la API, los parámetros que aceptan, los métodos HTTP permitidos y la estructura de los datos de solicitud y respuesta. Esto se hace utilizando un documento en formato JSON o YAML que sigue la especificación de OpenAPI.

- **Estándar y Legibilidad:** Al utilizar OpenAPI, las APIs se describen de una manera coherente y fácil de entender. Esto facilita que los desarrolladores comprendan cómo interactuar con la API sin necesidad de leer extensa documentación.
- **Generación de Documentación Automática**: Con la especificación de OpenAPI, es posible generar automáticamente documentación interactiva para la API, incluyendo una interfaz de usuario que permite probar los endpoints directamente desde el navegador.
- **Validación de la API:** Las herramientas compatibles con OpenAPI pueden realizar validaciones automáticas del cumplimiento de la especificación, lo que ayuda a detectar posibles errores en la implementación de la API.
- **Compatibilidad con Herramientas y Frameworks:** OpenAPI es ampliamente compatible con diversas herramientas y frameworks, lo que permite su integración con diferentes tecnologías y entornos de desarrollo.

22. Gracias a la extensión drf spectacular que hemos instalado y configurado al inicio de esta guía, generar la documentación de la aplicación REST de la aplicación *auctions* es muy sencillo. Ejecuta el siguiente comando:

    - `py manage.py spectacular --color --file schema.yml`

    Esto nos generará en la raíz de la carpeta, a la altura de manage.py, un fichero llamado **schema.yml** con la documentación (siguiendo OpenAPI 3). Si lo abrimos:

```yaml
openapi: 3.0.3
info:
  title: API Auctions
  version: 1.0.0
  description: Auctions web
paths:
  /api/auctions/:
    get:
      operationId: api_auctions_list
      parameters:
      - name: page
        required: false
        in: query
        description: A page number within the paginated result set.
        schema:
          type: integer
      tags:
      - api
      security:
      - cookieAuth: []
      - basicAuth: []
      - {}
      responses:
        '200':
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/PaginatedAuctionListCreateList'
          description: ''
```

- El nodo **info** corresponde con lo especificado en el fichero myFirstApiREST/setting.py en la variable **SPECTACULAR_SETTINGS.**
- En **paths** se definen cada uno de los endpoints del proyecto, es decir, de todas las aplicaciones. Para la url "/api/auctions/" se define su comportamiento según el método get (lista de subastas) y post (crear nueva subasta).
  Además, se incluye los parámetros si los hubiese (nombre, tipo y descripción). En este caso se infiere el parámetro page de la paginación de las listas de resultados.
  También se indica los códigos de estados que puede devolver el servidor.

Swagger

Swagger es un conjunto de herramientas de código abierto que permite diseñar, construir, documentar y consumir APIs RESTful de manera eficiente y colaborativa.

Originalmente, Swagger era el nombre de un proyecto de código abierto creado por SmartBear Software, pero posteriormente se renombró como "OpenAPI Specification". Por eso es habitual que la gente use Swagger y OpenApi como sinónimos, pero no lo son.

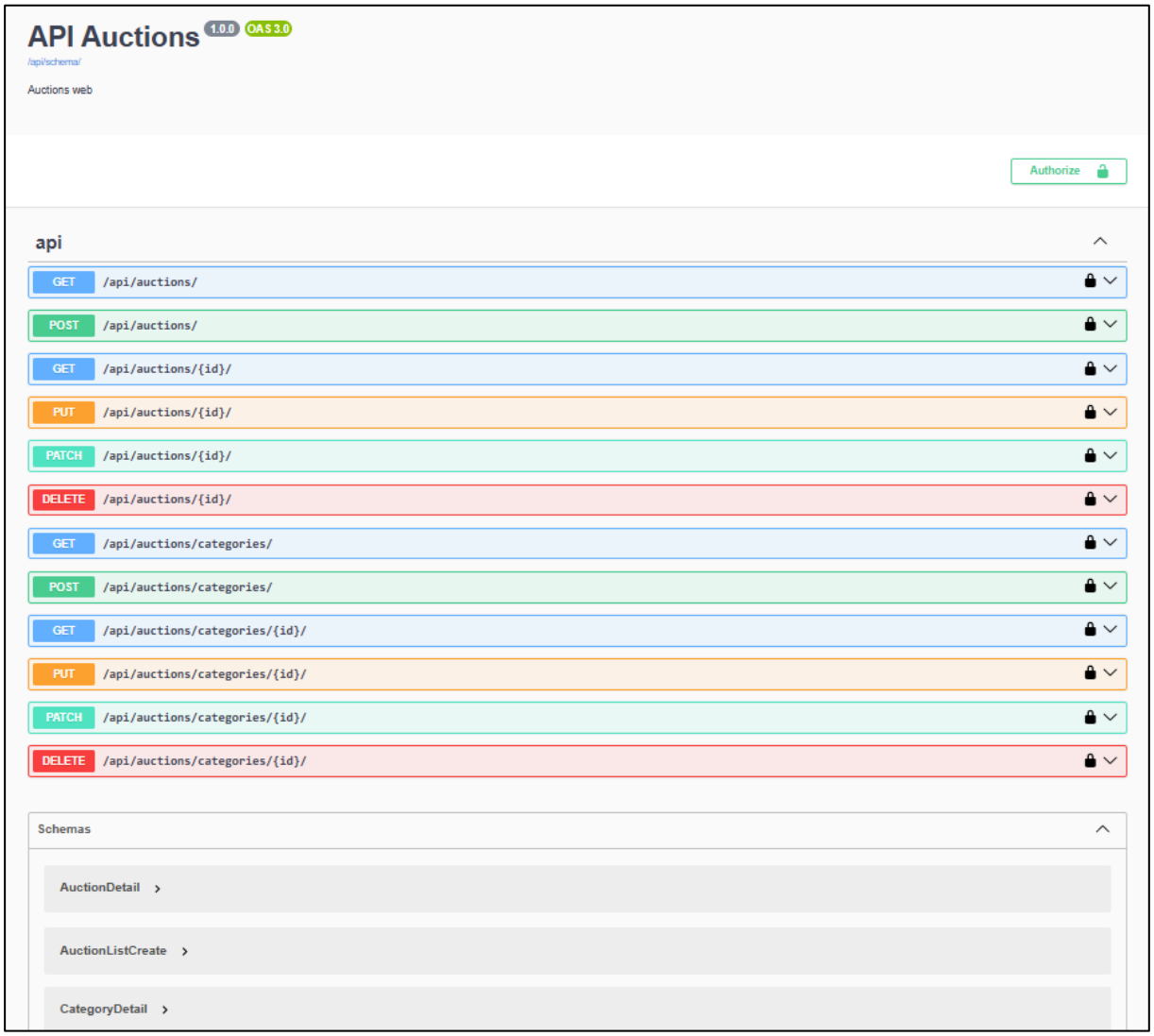
Swagger es un conjunto de herramientas para el diseño y documentación de APIs RESTful, mientras que OpenAPI es la especificación subyacente que define cómo describir APIs de manera estándar. Swagger se basa en la especificación OpenAPI y proporciona herramientas que permiten trabajar con esta especificación de manera efectiva.

23. Accede a myFirstApiRest/urls.py para añadir el swagger a tu proyecto django.

```
from django.contrib import admin
from django.urls import include, path
from drf_spectacular.views import SpectacularAPIView,
SpectacularSwaggerView

urlpatterns = [
    path("api/auctions/", include("auctions.urls")),
    path("admin/", admin.site.urls),
    path('api/schema/', SpectacularAPIView.as_view(), name='schema'),
   path('api/schema/swagger-ui/', SpectacularSwaggerView.as_view(url_name='schema'),
name='swagger-ui'),
]
```

24. Accede a http://127.0.0.1:8000/api/schema/swagger-ui/



Si pinchas en algunas de las secciones de colores verás la descripción del servicio; es decir, el método, la urls y los parámetros necesarios para realizar la respuesta. Y, además, podrás ver los ejemplos de respuesta que ofrecen. Incluso podrás realizar la petición, como si fuese un Postman.

Al final del todo, en Schemas, se encuentran los modelos de datos. Observa el campo “isOpen” de AuctionDetail. El sistema automático no ha sido capaz de inferir el tipo correcto de isOpen (booleano) y le ha seteado el por defecto: string.

Este problema se nos advertía en los logs tras ejecutar el comando: ***py manage.py spectacular --color --file schema.yml***

```
Warning [AuctionRetrieveUpdateDestroy > AuctionDetailSerializer]: unable to resolve type hint for function "get_isOpen". Consider using a type hint or @extend_schema_field. Defaulting to string.
```

Para solucionarlo, seguiremos el consejo de los logs y emplearemos el decorador "@extend_schema_field". Por tanto, añade (dejando lo ya existente) las siguientes líneas marcadas en negrita en el fichero auctions/serializers.py.

```
from drf_spectacular.utils import extend_schema_field
# Resto de imports

#Resto de serializadores
class AuctionListCreateSerializer(serializers.ModelSerializer):
    #Definición del serializador

    @extend_schema_field(serializers.BooleanField())
    def get_isOpen(self, obj):
        return obj.closing_date > timezone.now()


class AuctionDetailSerializer(serializers.ModelSerializer):
    #Definición del serializador

    @extend_schema_field(serializers.BooleanField())
    def get_isOpen(self, obj):
        print(obj.closing_date)
        print(timezone.now())
        return obj.closing_date > timezone.now()
```

A continuación, elimina el schema.yml actual y vuélvelo a generar usando el comando:

- `py manage.py spectacular --color --file schema.yml`

Arranca de nuevo el servidor:

- `py manage.py runserver`

Accede a http://127.0.0.1:8000/api/schema/swagger-ui/

Observa que *isOpen* ahora ya tiene el tipo correcto.

Ejercicio 7

Crea el api para realizar las operaciones CRUD sobre las pujas (**bid**). El nuevo modelo de pujar deberá estar formado por los siguientes campos:

- **id**: identificador propio de las pujas. *Numérico*.
- **auction**: la subasta a la que hace referencia. *Relación N-1.*
- **price**: cantidad ofrecida en la puja. *Numérico con dos decimales.*
- **creation_date**: fecha en la que se ha creado la puja. *Fecha auto generada.*
- **bidder**: nombre de usuario del postor. *Cadena de caracteres.*

Como se indica en *id_auction*, la puja tiene una relación N-1 con las subastas. Es decir, una puja se realiza sobre una subasta en concreto. Pero una subasta tendrá muchas pujas.

Se deberá implementar el modelo, los serializadores, las vistas y las urls. Además, se deberá informar la base de datos (mediante la shell de django) con unos ejemplos de pujas.

Las rutas de las pujas serán:

- /api/auctions/id_auctions/bid: Para el listado y la creación de pujas, donde *id_auctions* es el identificador de una subasta en concreto.
- /api/auctions/id_auctions/bid/id_bid: Para consultar, editar o borrar una puja, donde *id_auctions* es el identificador de una subasta en concreto y *id_bid* el de una puja específica.

Pista: para poder acceder a los *query params* (**auction_id o id**) en las vistas puedes usar **self.kwargs['auction_id']** según se explica aquí.

Pista 2: Recuerda ejecutar los 2 comandos de las migraciones tras la creación del modelo Bid.