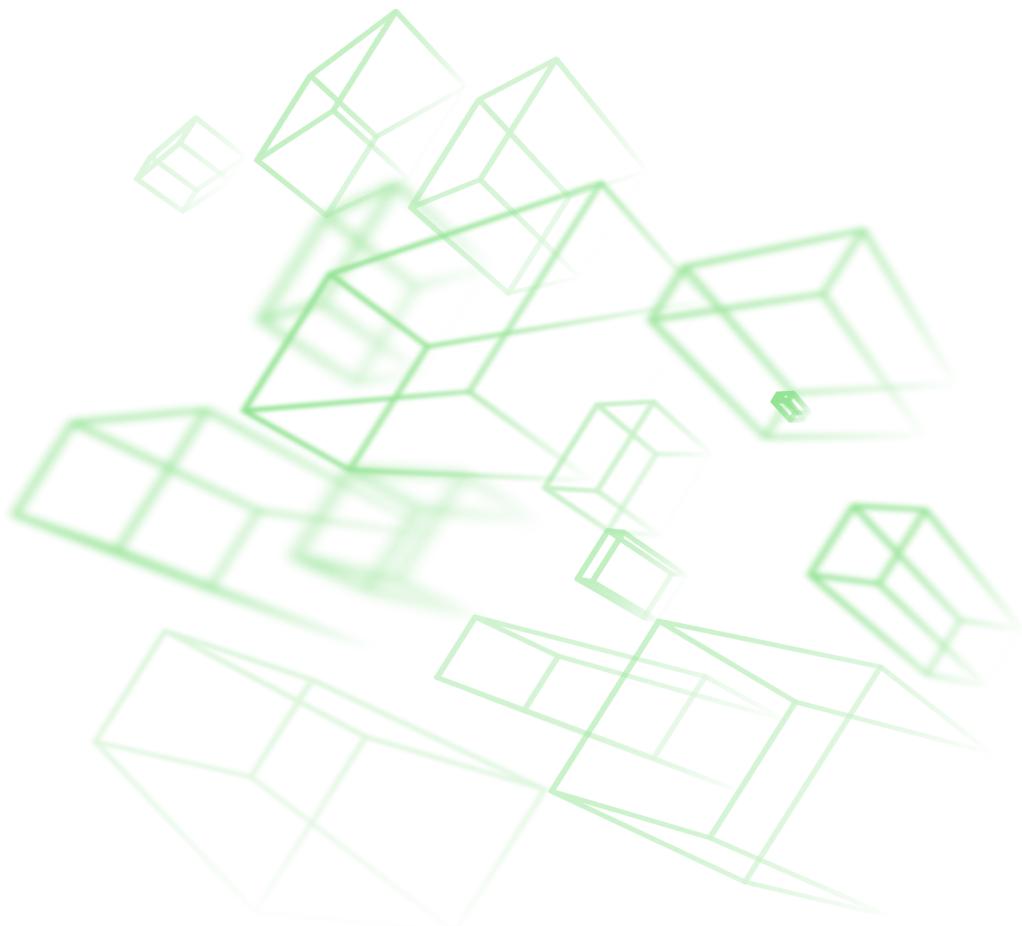




Apuntes de clase:

Python Django: REST FRAMEWORK 2



Profesores: Hector Vergara, Hernan Contigiani

Tabla de contenido

¿Qué es Django REST FRAMEWORK (DRF)?	2
Instalación de REST FRAMEWORK	2
Instalación en docker	3
Otro método	3
Declarando DRF dentro de Django	4
¿Dónde colocamos nuestras APIs?	5
Pipeline de armado de una API (genérica)	5
Serializadores	5
Creando nuestro primer serializador	6
Ejemplo práctico	7
Manejo de foreign keys en serializadores	7
Construcción de las vistas de API (API views)	9
Vistas genéricas de DRF	10
Niveles de acceso con TOKENs	11
Generando las URLs para nuestras APIs	11
Mensajes HTTP: una mirada más cercana	12
Header	13
Body	14
TOKENs en mensajes HTTP	14
Del lado del cliente	14
Del lado del servidor	15
Generando TOKENs en Django Admin	15
Generando TOKENs desde una API	16
Haciendo el código	16
Asignamos una url a nuestra API de logueo	20
En resumen...	21
Swagger	22
Instalando dependencias en nuestro entorno/sistema	22
Descargar un repositorio de GitHub	27
Links de interés	27

¿Qué es Django REST FRAMEWORK (DRF)?

Django REST FRAMEWORK es un framework para el desarrollo de APIs dentro del ecosistema de Django.

 [Página oficial](#)

Este framework se ha hecho muy popular entre los desarrolladores por lo completo y fácil de utilizar, en él vamos a encontrar una guía de cómo desarrollar nuestras APIs de manera estandarizada y segura.

Las utilidades que vamos a desarrollar son:

- ⚡ **Decoradores** para implementar APIs rápidas.
- ⚡ **TOKENS** para darle un grado de seguridad al acceso a nuestros recursos de servidor.
- ⚡ **Serializadores** para el manejo de los datos desde y hacia la base de datos.
- ⚡ **API views** (vistas de API) como espacio para desarrollar la lógica en nuestras APIs.
- ⚡ **Interface web** para la prueba y utilización de nuestras APIs.

Instalación de REST FRAMEWORK

Para la instalación de DRF, tenemos que instalar la librería en nuestro sistema, para nuestro caso vamos a declararlas en nuestro archivo de **requirements.txt**:

```
djangorestframework==3.12.4  
django-rest-auth==0.9.5  
django-filter==2.4.0
```

Luego tenemos que correr el comando

```
pip install -r requirements.txt
```

O bien instalar cada una con el gestor de paquetes PIP:

```
pip install djangorestframework==3.12.4
```

```
pip install django-rest-auth==0.9.5
```

```
pip install django-filter==2.4.0
```

Instalación en docker

Para instalarlo en la imagen preexistente de Docker, solo basta con incluirlo dentro del archivo requirements.txt y utilizar los siguientes comandos en una consola en el sistema host:

Detenemos el contenedor que está funcionando. Para ello dentro de la carpeta que contiene el archivo docker-compose.yml hacemos:

```
$ docker-compose stop
```

Luego, re-compilamos la imagen a partir del archivo dockerfile presente en el mismo directorio:

```
$ docker-compose build
```

Y por último, una vez terminado el proceso, iniciamos los servicios:

```
$ docker-compose up
```

De esta manera hacemos un "rebuild" de la imagen que ya existía y actualizamos el contenido de las dependencias.

Otro método

Siendo que dentro del contenedor de software está corriendo un sistema operativo linux, podríamos instalar las nuevas dependencias utilizando los comandos del gestor de paquetes de PIP.

Para ello tenemos que ingresar dentro del contenedor que debe estar corriendo con el comando:

```
$ docker exec -i -t [NOMBRE_DEL_CONTENEDOR] bash
```

Este conocido comando nos va a devolver una consola en donde tendremos que ejecutar los comandos antes mencionados:

```
pip3 install djangorestframework==3.12.4
```

```
pip3 install django-rest-auth==0.9.5
```

```
pip3 install django-filter==2.4.0
```



Importante: Con este método, las librerías **SÓLO SE INSTALAN EN EL CONTENEDOR NO EN LA IMAGEN**, esto quiere decir que si eliminamos dicho contenedor, tendremos que repetir la operación en cada contenedor que lancemos.

Declarando DRF dentro de Django

Al igual que el resto de las aplicaciones, DRF debe declararse en `settings.py` en la variable `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Local apps: Acá ponemos el nombre de las carpetas de nuestras aplicaciones
    'e_commerce',
    # Third party apps: acá vamos agregando las aplicaciones de terceros, extensiones de Django.
    'rest_framework',
    'rest_framework.authtoken',
]
```

Cómo vemos, en los últimos dos índices de nuestra lista se encuentran:

```
'rest_framework',
'rest_framework.authtoken',
```

También es necesario colocar un nuevo diccionario dentro de `settings.py` que será de uso exclusivo de DRF:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework.authentication.TokenAuthentication',
        'rest_framework.authentication.SessionAuthentication',
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
    'DEFAULT_FILTER_BACKENDS': ['django_filters.rest_framework.DjangoFilterBackend']
}
```

Luego de realizar estos cambios, debemos realizar una migración del sistema, ya que DRF agrega modelos en nuestra base de datos, en los cuales guarda entre otras cosas, los TOKENs que vayamos generando para cada usuario.

Todo esto está especificado en el tutorial de instalación de DRF.

 [Tutorial de instalación de DRF](#)

¿Dónde colocamos nuestras APIs?

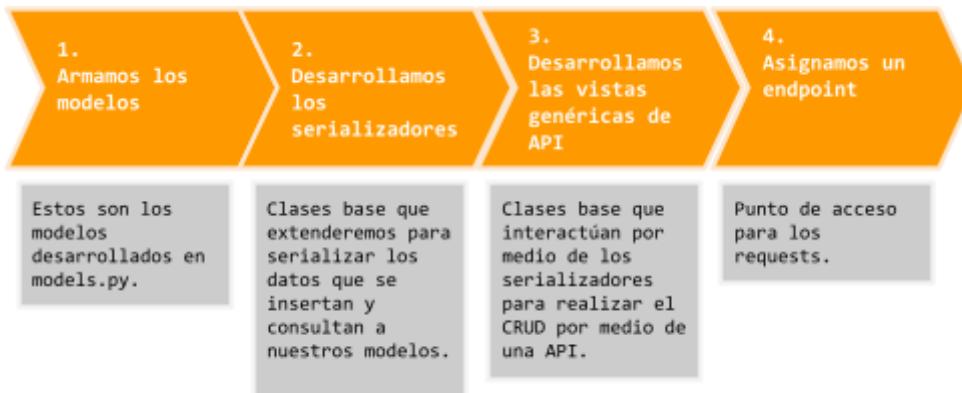
Para la arquitectura de carpetas, recomendamos seguir la guía Django REST FRAMEWORK 1, en donde se aborda la siguiente estructura con más detalle:

- Estructura inicial de nuestra API -

- 📁 **mi_proyecto** → Directorio raíz
- └ 📁 **aplicación** → Aplicación.
 - └ 📁 **api** → **Carpeta de APIs**
 - └ 📜 **archivo.py** → Archivo de lógica de API

Pipeline de armado de una API (genérica)

Hay muchas formas de desarrollar nuestra API, hemos visto una forma de organizarnos en el documento anterior [DJANGO REST FRAMEWORK 1] para poder hacerlo. En este caso, vamos a extender ese pipeline agregando una etapa más:



Serializadores

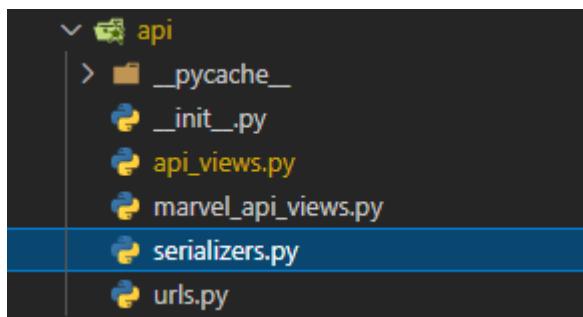
Los serializadores son los que se encargan de transformar el formato de los datos desde el request, que llegan en formato JSON, hasta los modelos de nuestras bases de datos. Recordemos que los campos de una entidad en las bases de datos pueden tomar diferentes formatos, como strings, números enteros, flotantes, datetime, formato email, texto largo, etc;

Dentro del JSON que llega en nuestro request, solo vamos a encontrar keys y values en formato numérico o strings, posiblemente dentro de un array.

Para poder lidiar con esta transformación de datos, DRF nos proporciona serializadores que se encargan de ello, de una manera sumamente sencilla y segura.

Creando nuestro primer serializador

Para ello primero crearemos un archivo python al que llamaremos "serializers.py" dentro del directorio de nuestras APIs:



Dentro de él, tenemos que importar tanto los modelos como los serializadores estándar de DRF. Para nuestro caso:

```
# Primero importamos los modelos que queremos serializar:  
from e_commerce.models import Comic,WishList  
from django.contrib.auth.models import User  
# Luego importamos todos los serializadores de django rest framework.  
from rest_framework import serializers
```

"serializers" es la clase base de la que vamos a importar "serializers.ModelSerializer" para extender las clases que construyen nuestros serializadores:

```
class NombreDelSerializador(serializers.ModelSerializer):  
    class Meta:  
        model = ModeloImportado  
        fields = ('__all__')
```

Con esto ya tenemos nuestro serializador terminado! Aunque solo de manera básica y siempre y cuando el modelo no contenga Foreign Keys.

Observemos que dentro de nuestra clase, a la cual le asignaremos un nombre por nuestra cuenta, tendremos una clase META en la cual debemos especificar dos atributos como mínimo:

- model: El modelo que vamos a serializar
- fields: los campos del modelo que vamos a incluir en el serializador. Este admite una tupla que contenga '__all__' únicamente, pero también puede desglosarse en los campos de nuestro modelo, como una tupla de strings. No es necesario que agreguemos el id.

Ejemplo práctico

```
# Primero importamos los modelos que queremos serializar:  
from e_commerce.models import Comic  
# Luego importamos todos los serializadores de django rest framework.  
from rest_framework import serializers  
  
class ComicSerializer(serializers.ModelSerializer):  
  
    class Meta:  
        model = Comic  
        fields = ('__all__')
```

O bien, de manera equivalente:

```
# Primero importamos los modelos que queremos serializar:  
from e_commerce.models import Comic  
# Luego importamos todos los serializadores de django rest framework.  
from rest_framework import serializers  
  
class ComicSerializer(serializers.ModelSerializer):  
  
    class Meta:  
        model = Comic  
        fields = ('marvel_id','title', 'description', 'price', 'stock_qty', 'picture')
```

Manejo de foreign keys en serializadores

Recordemos que un foreign key representa la relación entre dos tablas. Entonces lo que necesita nuestro serializador es pasarle dentro de los campos (**fields**) un objeto del modelo a relacionar, ya que si vamos a hacer un "CREATE" tenemos que insertar el registro en ambos modelos, o bien utilizar un registro preexistente para nuestro FK.

Para hacer esto, dentro de nuestra clase principal, tenemos que declarar un nuevo atributo con el nombre del campo (**field**) correspondiente al **FK**, y llamar al método **serializers.PrimaryKeyRelatedField()**:

```
campo_relacionado = serializers.PrimaryKeyRelatedField(write_only=True,
                                                       queryset=modelo_relacionado.objects.all())
```

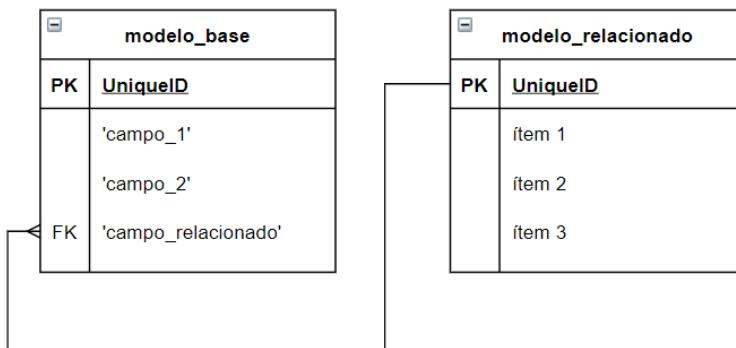
En donde:

- **campo_relacionado**: es el nombre que le dimos en nuestra tabla al campo que está relacionado y utiliza el `models.ForeignKey()`
- **write_only**: Nos permite insertar en la entidad.
- **queryset**: Debemos especificar una consulta al modelo relacionado para poder leer los datos del mismo de ser necesario.

Una vez hecho esto, nos debería quedar algo similar a esto:

```
class PruebaSerializer(serializers.ModelSerializer):
    campo_relacionado = serializers.PrimaryKeyRelatedField(write_only=True,
                                                           queryset=modelo_relacionado.objects.all())
    class Meta:
        model = modelo_base
        fields = ['campo_1', 'campo_2', 'campo_relacionado']
```

De esta manera **modelo_base** es la primera entidad que contiene el **FK** de otra entidad a la que llamamos "**modelo_relacionado**".



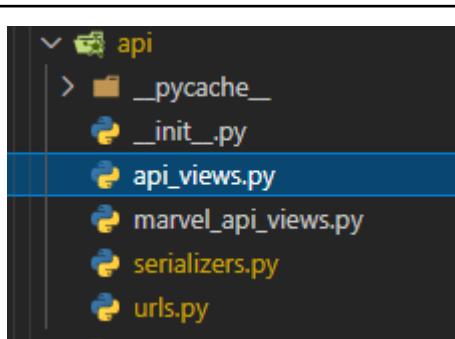
Nota: vemos que "**campo_relacionado**" es el nombre que toma el atributo en la entidad "**modelo_base**"!

 [Documentación en DRF de PrimaryKeyRelatedField](#)

Construcción de las vistas de API (API views)

Para mantenernos dentro del patrón de diseño MVT, vamos a trabajar la lógica de nuestras APIs en lo que denominaremos "vistas de API" o "API views".

Existen muchas formas de construir APIs, es por ello que vamos a tratar de estandarizar su construcción.



Siguiendo el "pipeline de armado de una API" este sería el tercer paso, y para ello, construiremos un archivo dedicado a esta tarea, dentro de nuestro directorio /api.

También, de ser necesario, podemos crear otros archivos de vista de api para propósitos específicos como es en nuestro caso de marvel_api_views.py.

Nuestro archivo api_views.py va a importar diferentes módulos para la construcción de nuestras APIs, entre ellos:

- 👉 Los serializadores creados en **serializers.py**
- 👉 Los modelos con los que vamos a trabajar (de **models.py**)
- 👉 Módulos de vistas genéricas de **DRF**
- 👉 Utilidades para la implementación de **TOKENs**
- 👉 Utilidades varias que vayamos necesitando.

Ejemplo de importación de modelos y serializadores:

```
# Primero, importamos los serializadores
from e_commerce.api.serializers import *

# Segundo, importamos los modelos:
from django.contrib.auth.models import User
from e_commerce.models import Comic, WishList
```

Vistas genéricas de DRF

Django REST framework nos proporciona una cantidad de módulos para esta construcción, cada una nos permite realizar una o más partes del CRUD.

Todas parten de la misma base:

```
from rest_framework.generics import [VISTA GENERICA]
```

Entre ellas, vamos a utilizar:

- **ListAPIView**: Nos proporciona un método **GET** para manejo de peticiones.
- **CreateAPIView**: Nos proporciona un método **POST** para el manejo de peticiones.
- **ListCreateAPIView**: Nos proporciona tanto **GET** como **POST**.
- **RetrieveUpdateAPIView**: Método **GET-PUT-PATCH**
- **DestroyAPIView**: Método **DELETE**

Se utilizan como base para extender clases, por ejemplo para el caso de ListAPIView:

```
class NombreDeAPIView(ListAPIView):  
    '''[METODO GET]  
    Aquí vamos a documentar nuestra API.  
    ...  
    queryset = ModeloImportado.objects.all()  
    serializer_class = SerializadorDelModeloImportado  
    permission_classes = []
```

Dentro de cada vista genérica debemos especificar:

- **queryset**: la consulta a la base de datos que va a realizar.
- **serializer_class**: El serializador adecuado para ese modelo.
- **permission_classes**: Los permisos de acceso (vacío para dejarlo con libre acceso)

Esta metodología se aplica a todas las vistas genéricas, así pues para RetrieveUpdateAPIView:

```
class OtroNombreDeAPIView(RetrieveUpdateAPIView):  
    '''[METODO GET-PUT-PATCH]  
    Aquí vamos a documentar nuestra API.  
    ...  
    queryset = ModeloImportado.objects.all()  
    serializer_class = SerializadorDelModeloImportado  
    permission_classes = []
```

Niveles de acceso con TOKENs

Para restringir el acceso a nuestras APIs vamos a utilizar el atributo **permission_classes** dentro de nuestras vistas genéricas. Para ello debemos importar las utilidades que nos permitan generar una jerarquía de acceso:

```
from rest_framework.permissions import IsAuthenticated, IsAdminUser
```

Estos dos atributos nos permiten:

- **IsAuthenticated**: El usuario debe estar autenticado para poder utilizar la API
- **IsAdminUser**: El usuario además de estar autenticado, debe ser administrador del sitio

Estos atributos se insertan dentro de nuestra lista **permission_classes**:

```
permission_classes = [IsAuthenticated, IsAdminUser]
```

Y de esta manera tenemos restringido el uso de nuestras APIs.

Generando las URLs para nuestras APIs

Esta es la última etapa de nuestro “Pipeline de desarrollo de APIs”, y es igual al detallado en el documento Django REST FRAMEWORK 1. Debemos incluir nuestros endpoints como path dentro de la lista de **urlpatterns**:

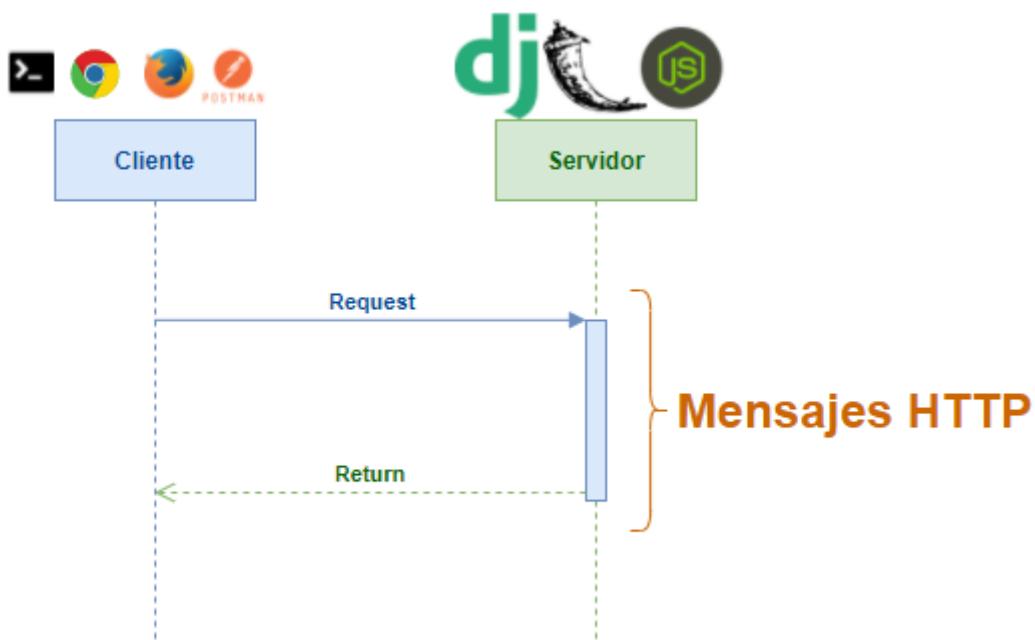
```
path('recurso/get', NombreDeAPIView.as_view()),
```

Desde luego que primero debemos importar nuestra vista de api, y luego la utilizaremos en path() con un método que poseen las vistas de api llamado **.as_view()**

Mensajes HTTP: una mirada más cercana

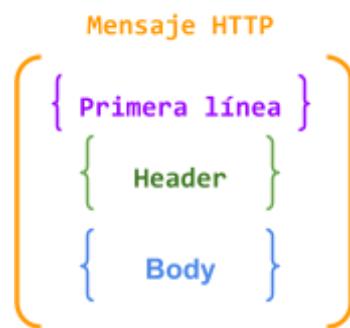
Antes de adentrarnos en el uso de TOKENS es importante entender cómo es que están compuestos los mensajes HTTP.

La comunicación entre clientes y servidores web se realiza por medio de mensajes HTTP siguiendo el siguiente comportamiento:



El cliente inicia la petición (request) a un recurso del servidor, el cual retorna un resultado esperado.

La estructura del mensaje HTTP tiene principalmente dos componentes: el **header** y el **body**:



Header

La **primera línea** determina el tipo de petición, si es GET, POST, PUT, etc; , luego comienza el **HEADER** y lo separa un espacio el body:

HTTP/1.x message



Dentro del header se envían los parámetros de configuración del mensaje, por ejemplo, el host al que va a realizar la petición, el tipo de contenido (Content-type), entre otros elementos que dependen de la petición.

Otro ejemplo de un header de una petición realizada con postman a nuestro servidor local:

```
▼ Request Headers
  Content-Type: "application/json"
  User-Agent: "PostmanRuntime/7.28.2"
  Accept: "*/*"
  Postman-Token: "d44866c3-74e4-48b5-be14-860d0321194a"
  Host: "localhost:8000"
  Accept-Encoding: "gzip, deflate, br"
  Connection: "keep-alive"
  Content-Length: "58"
  Cookie: "csrfToken=cfEuCX6qThpN6UC9eXypC71j6A4KJQagRSojPnqXfZjN5wJg09hXXQKCU8Vf1LDR"
```

Las librerías como "requests" en python nos permiten abstraer de tener que especificar cada elemento en una petición

Body

El body (cuerpo) del mensaje, es el que contiene el contenido del mensaje que queremos enviar/recibir.

Para nuestro caso, en el cual vamos a trabajar con APIs, la información que vamos a enviar está en formato JSON. Un ejemplo en POSTMAN del body de un request:

```
{  
  "username": "root",  
  "password": "12345"  
}
```

 **Importante:** Para poder enviar información en formato JSON, debemos establecer el **Content-Type:"application/json"** en el header de nuestro mensaje.

TOKENs en mensajes HTTP

Los TOKENs son una clave alfanumérica que viaja en el header de nuestro mensaje HTTP.

Del lado del cliente

Existen distintos tipos de TOKENs, los que utilizaremos en Django REST Frameworks nos ayudarán en el proceso de autenticación, es por ello que debemos declararlos en el header con la key "Authorization" y el value (por ejemplo) "Token num3r0d3t0k3n":

```
▼ Request Headers  
  authorization: "Token 92937874f377a1ea17f7637ee07208622e5cb5e6"  
  User-Agent: "PostmanRuntime/7.28.2"  
  Accept: "*/*"  
  Postman-Token: "db8bfc09-1074-4bb9-9098-098481edcb86"  
  Host: "localhost:8000"  
  Accept-Encoding: "gzip, deflate, br"  
  Connection: "keep-alive"  
  Cookie: "csrftoken=cfEuCX6qThpN6UC9eXypC71j6A4KJ0agRSojPnaXfZjN5wJg09hXXOKCU8VfLLDR"
```

Esto es para el proceso de REQUEST de parte del cliente, es decir, el cliente debe tener de antemano el TOKEN para poder realizar la consulta a un recurso protegido.

Del lado del servidor

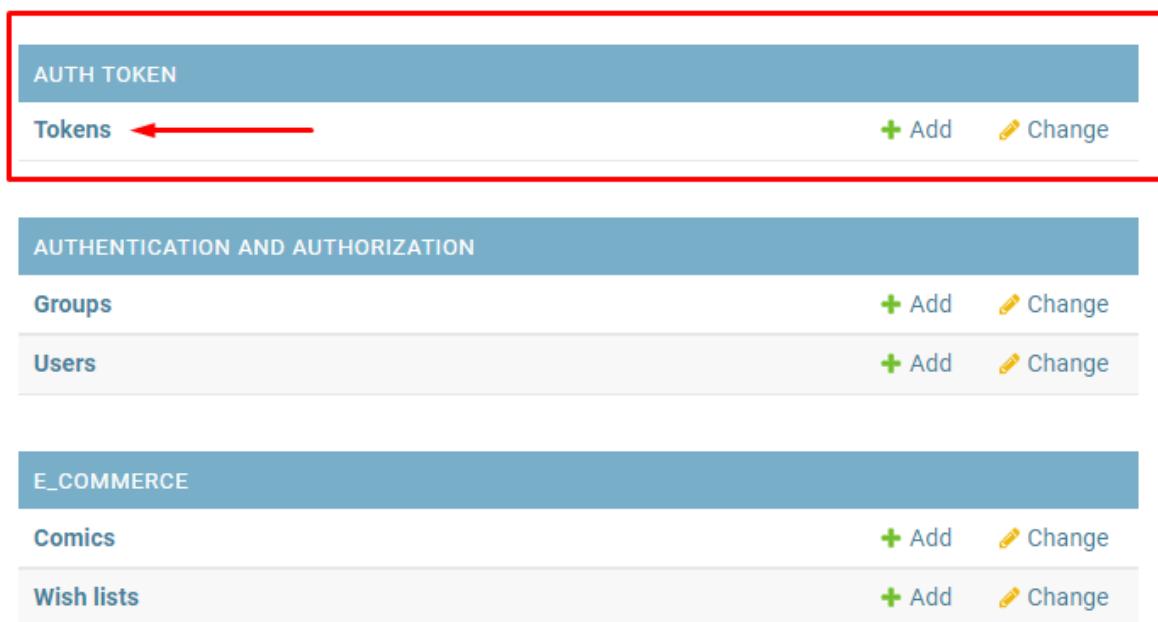
Del lado del servidor, los TOKENs se almacenan en la base de datos, y dependiendo del criterio que requiera la aplicación, pueden ser temporales (por sesión) o permanentes. Nosotros trabajaremos con TOKENs permanentes para el propósito de este curso. Cada usuario posee un TOKEN único e intransferible, el cual podemos generar de dos maneras diferentes:

- Por medio de una interfaz gráfica, manualmente
- Por medio de una petición, el usuario al momento de autenticarse en una API, se le devuelve su TOKEN de sesión.

Generando TOKENs en Django Admin

Al iniciar sesión en el administrador de Django, debemos ir a nuestra aplicación "Auth Token" y de allí a "Tokens".

Site administration



The screenshot shows the Django Admin dashboard with three main sections:

- AUTH TOKEN**: Contains a table with one row labeled "Tokens" and buttons for "+ Add" and "Change". A red arrow points to the "Tokens" link.
- AUTHENTICATION AND AUTHORIZATION**: Contains tables for "Groups" and "Users", each with "+ Add" and "Change" buttons.
- E_COMMERCE**: Contains tables for "Comics" and "Wish lists", each with "+ Add" and "Change" buttons.

Allí se encontrarán listados los tokens existentes y podremos crear un nuevo token si hacemos click en "ADD TOKEN":

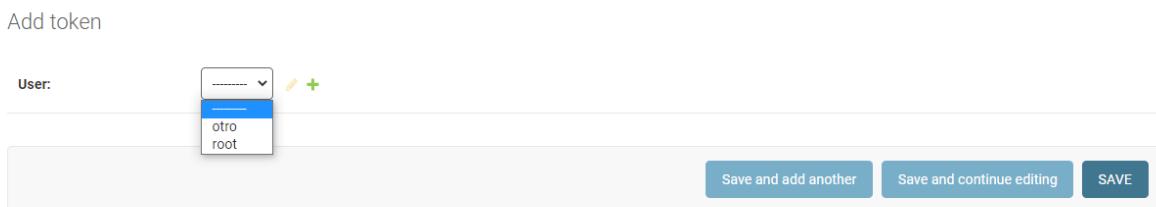


The screenshot shows the "Tokens" list view in the AUTH TOKEN section of the Django Admin. It includes a header with "Select token to change" and an "ADD TOKEN" button with a red arrow pointing to it. The table displays two tokens:

KEY	USER	CREATED
92937874f377a1ea17f7637ee07208622e5cb5e6	root	July 16, 2021, 10:48 p.m.
7c8ca8efd2fb1a6d6588c142d78d0375b8101a57	otro	July 24, 2021, 6:44 a.m.

Below the table, it says "2 tokens".

Allí simplemente debemos seleccionar un usuario al cual le asignaremos dicho token y presionaremos SAVE:



Generando TOKENs desde una API

Este proceso es más complejo, pero al mismo tiempo nos da la ventaja de ser automatizado, ya que si el usuario hace una petición para autenticarse, si no existe su TOKEN haremos que se genere automáticamente.

Para ello vamos a crear una vista de API genérica-personalizada, esto es posible tomando una vista de API genérica que nos proporciona DRF y vamos a hacer un "override" de su función "**post()**"

Este nuevo **post()** va a cumplir los siguientes requisitos:

- 👉 Requiere un JSON de entrada que especifique el username y password del usuario
- 👉 Realiza el proceso de autenticación: Verifica que el username y el password existan en la base de datos y sean correctos.
- 👉 Verifica qué TOKEN tiene asignado el usuario, de no contar con uno, lo debe crear.
- 👉 Armar el JSON de respuesta con los datos de usuario + el TOKEN correspondiente.

Haciendo el código

Primero importamos de DRF la clase "APIView" que a diferencia de las anteriores, es una clase base que posee los métodos de **get()** y **post()** implementados y listos para ser modificados.

```
from rest_framework.views import APIView
```

Más info:

 [APIView](#)

También vamos a importar un parseador JSON de DRF que nos permitirá recibir peticiones de **Content-Type: "application/json"**:

```
from rest_framework.parsers import JSONParser
```

Más info:

 [Seteo de parceadores \[JSON PARSER\]](#)

Entonces, en nuestro código:

```
class LoginUserAPIView(APIView):
    ...
    Vista de API personalizada para recibir peticiones de tipo POST.
    Esquema de entrada:
    {"username":"root", "password":12345}

    Utilizaremos JSONParser para tener 'Content-Type': 'application/json'
    ...
    parser_classes = [JSONParser]
    authentication_classes = []
    permission_classes = []
```

Vemos que para que nuestra API trabaje con mensajes JSON en su body tenemos que ajustar el atributo:

```
parser_classes = [JSONParser]
```

También dejamos libre el acceso, ya que el usuario no tiene el TOKEN en esta instancia. Ahora debemos sobreescribir nuestro método **post()**, para ello, debemos definir su función dentro de nuestra clase:

```
class LoginUserAPIView(APIView):
    ...
    Vista de API personalizada para recibir peticiones de tipo POST.
    Esquema de entrada:
    {"username":"root", "password":12345}

    Utilizaremos JSONParser para tener 'Content-Type': 'application/json'
    ...
    parser_classes = [JSONParser]
    authentication_classes = []
    permission_classes = []

    def post(self, request, format=None):
```

Vemos que dentro de **post()** debe recibir

- 👉 **self**: porque es un método de una clase
- 👉 **request**: es la variable que va a traer los datos del mensaje HTTP
- 👉 **format=None**: requerido para que el body reciba un JSON ([VER DOCUMENTACIÓN EN DRF](#))

Dentro de la función primero, siempre, agregamos la documentación de este POST, luego dentro de un try and except, hacemos la lógica:

```
def post(self, request, format=None):
    ...
    Esta función sobrescribe la función post original de esta clase,
    recibe "request" y hay que setear format=None, para poder recibir los datos en request.data
    la idea es obtener los datos enviados en el request y autenticar al usuario con la
    función "authenticate()", la cual devuelve el estado de autenticación.
    \nLuego con estos datos se consulta el Token generado para el usuario, si no lo tiene
    asignado, se crea automáticamente.
    \nEsquema de entrada:
    \n`{"username": "root", "password": "12345"}`
    \nUtilizaremos JSONParser para tener `Content-Type: 'application/json'`
    ...
    user_data = {}
    try:
        # Obtenemos los datos del request:
        username = request.data.get('username')
        password = request.data.get('password')
        # Obtenemos el objeto del modelo user, a partir del usuario y contraseña,
        # NOTE: es importante el uso de este método, porque aplica el hash del password!
        account = authenticate(username=username, password=password)

        if account:
            # Si el usuario existe y sus credenciales son validas, tratamos de obtener el TOKEN:
            try:
                token = Token.objects.get(user=account)
            except Token.DoesNotExist:
                # Si el TOKEN del usuario no existe, lo creamos automáticamente:
                token = Token.objects.create(user=account)
            # Con todos estos datos, construimos un JSON de respuesta:
            user_data['user_id'] = account.pk
            user_data['username'] = username
            user_data['first_name'] = account.first_name
            user_data['last_name'] = account.last_name
            user_data['email'] = account.email
            user_data['is_active'] = account.is_active
            user_data['token'] = token.key
            # Devolvemos la respuesta personalizada
            return Response(user_data)
        else:
            # Si las credenciales son invalidas, devolvemos algun mensaje de error:
            user_data['response'] = 'Error'
            user_data['error_message'] = 'Credenciales invalidas'
            return Response(user_data)

    except Exception as error:
        # Si aparece alguna excepción, devolvemos un mensaje de error
        user_data['response'] = 'Error'
        user_data['error_message'] = error
        return Response(user_data)
```

Para que esta lógica funcione, necesitamos importar adicionalmente:

- 👉 El modelo “**Token**”, para consultar sus registros en la base de datos:

```
from rest_framework.authtoken.models import Token
```

- 👉 El método **authenticate()** necesario para comprobar el **username** y **password**
(Recordemos que el password está hasheado en la base de datos, por ello, no podemos utilizar un filtro ordinario!):

```
from django.contrib.auth import authenticate
```

- 👉 El método **Response()** nos permite construir una respuesta http recibiendo un diccionario como parámetro.

```
from rest_framework.response import Response
```

Luego la lógica de este proceso es simple:

1. Recolectamos la información que necesitamos desde el **request.data.get()**
2. Con la información disponible, autenticamos al usuario con **authenticate()**, este método nos retorna el objeto **User**.
3. Realizamos una query en la entidad “**Token**” para ver si el usuario tiene su **Token**, de no ser así, creamos un token con **Token.objects.create(user=account)** el argumento “**user**” recibe un objeto “**User**” que es el que obtuvimos en el punto 2, si el objeto no existe (porque no existe ese **User**, ya sea por un error en el password u otra cosa) lanza una excepción que devolveremos en el **Response()**
4. Una vez obtenido el TOKEN, construimos un diccionario al que llamaremos **user_data** con los datos del usuario + TOKEN.
5. Una vez construido el diccionario, lo retornamos con el método **Response()**

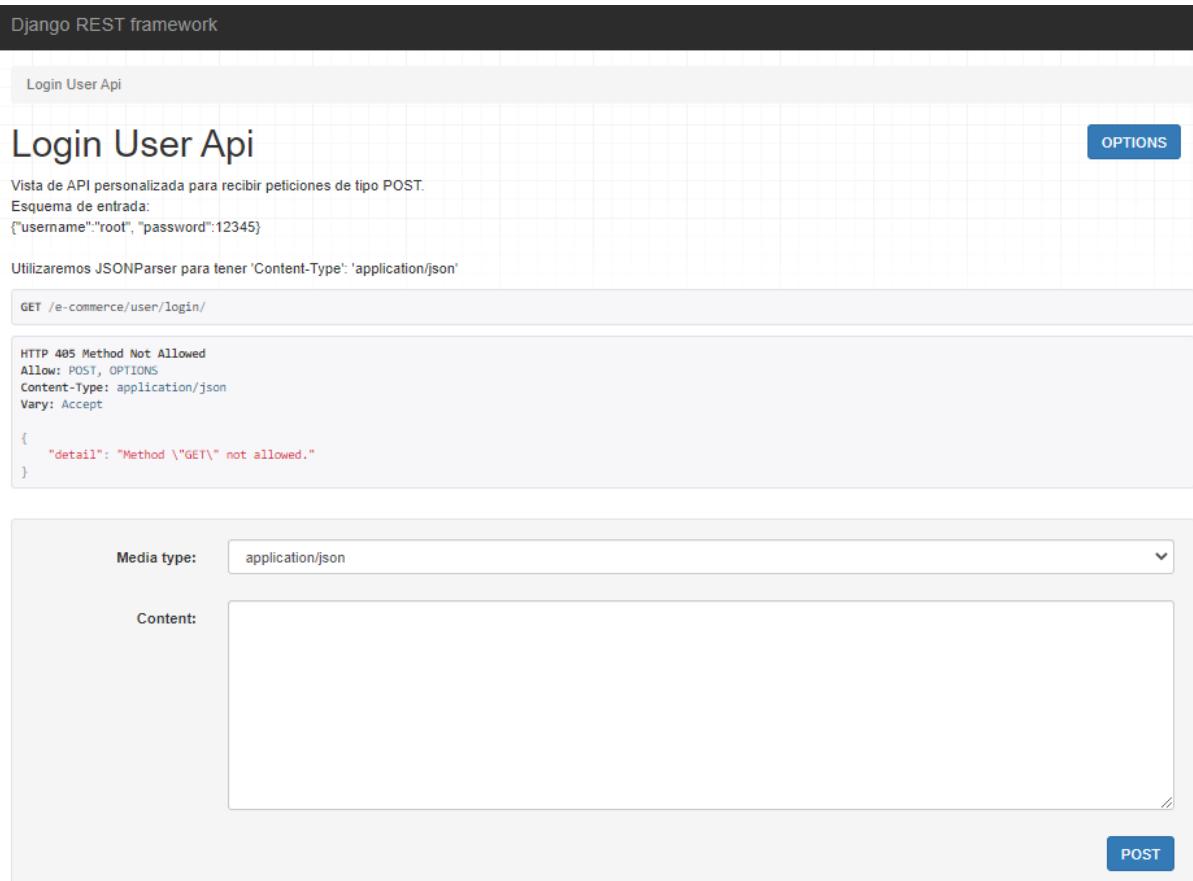
Asignamos una url a nuestra API de logueo

Dentro de nuestro archivo urls.py correspondiente a nuestras APIs, debemos importar nuestra nueva clase y posteriormente en nuestras urlpatterns, agregaremos:

```
urlpatterns = [
    # User APIs:
    path('user/login/', LoginUserAPIView.as_view()), ...]
```

La denominación de nuestros endpoints son arbitrarias.

Si navegamos en nuestro explorador en la dirección de este recurso nos devolverá:



The screenshot shows the Django REST framework's browsable API documentation for a 'Login User Api'. At the top, there's a header bar with 'Django REST framework' and a 'OPTIONS' button. Below it, the title 'Login User Api' is displayed, along with a note that it's a 'Vista de API personalizada para recibir peticiones de tipo POST.' It also specifies the 'Esquema de entrada' as a JSON object: {"username": "root", "password": "12345"} and notes that 'Utilizaremos JSONParser para tener 'Content-Type': 'application/json''.

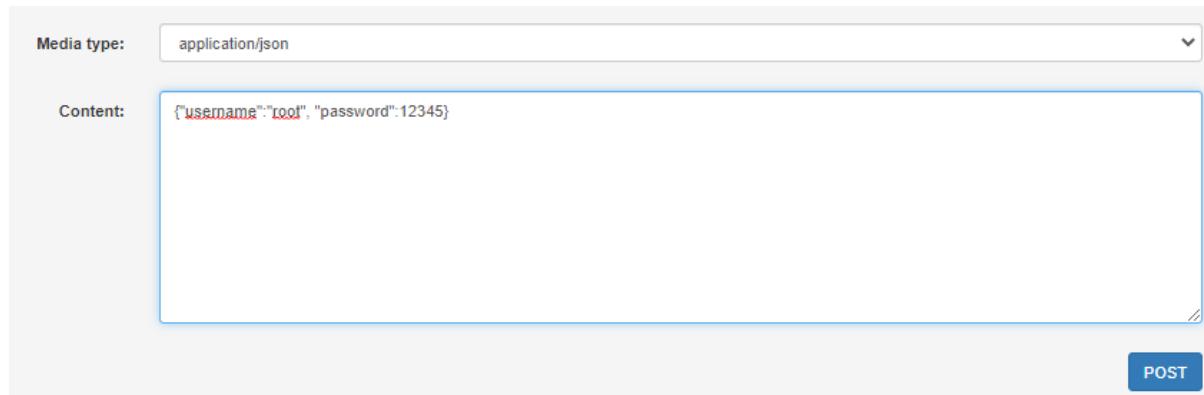
Below this, there's a section for a 'GET /e-commerce/user/login/' request, which returns an error message: 'HTTP 405 Method Not Allowed' with 'Allow: POST, OPTIONS' and a detailed error message: '{ "detail": "Method \"GET\" not allowed." }'.

At the bottom, there's a form for making a 'POST' request. It has a 'Media type:' dropdown set to 'application/json' and a large 'Content:' text area. A 'POST' button is located at the bottom right of the form.

Como vemos, el formato admitido es "application/json" y el campo del contenido nos permite ingresar un JSON a mano.

También es importante destacar que la documentación de esta API es la que colocamos al documentar la clase "class LoginUserAPIView(APIView)" y esto le permite al usuario que consuma este recurso, entender cómo es que funciona nuestra API.

Si brindamos el JSON de ejemplo (y los datos son reales, es decir, existe en la base de datos) podemos probar nuestra API:



Media type: application/json

Content: {"username": "root", "password": "12345"}

POST

Al lanzar el POST nos devolverá el siguiente mensaje:

```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "user_id": 1,
    "username": "root",
    "first_name": "",
    "last_name": "",
    "email": "hvergara@inove.com.ar",
    "is_active": true,
    "token": "92937874f377alea17f7637ee07208622e5cb5e6"
}
```

En resumen...

Ésta no es la única manera de realizar la autenticación en una API, pero sí una sencilla y efectiva, es importante también tener en cuenta los métodos vistos para otras aplicaciones.

Swagger

Swagger es una popular aplicación que se utiliza para la documentación de APIs. Y funciona para muchos frameworks, incluyendo a Django.

Sitio oficial: [Swagger API documentation site](#)

Para poder implementarlo en Django, debemos seguir unos simples pasos:

Instalando dependencias en nuestro entorno/sistema

Las dependencias que requiere Swagger son:

```
# Swagger:  
uritemplate==3.0.1  
pyyaml==5.4.1
```

Obviamente también debe estar previamente instalada la suite de Django REST Framework. Estas dependencias las podemos incluir en nuestro **requirements.txt** o simplemente instalarlas vía **PIP** cada una de ellas.

Una vez instalados, vamos a `settings.py` y configuramos el `REST_FRAMEWORK`:

```
REST_FRAMEWORK = {  
    'DEFAULT_AUTHENTICATION_CLASSES': (  
        'rest_framework.authentication.TokenAuthentication',  
        'rest_framework.authentication.SessionAuthentication',  
    ),  
    'DEFAULT_PERMISSION_CLASSES': (  
        'rest_framework.permissions.IsAuthenticated',  
    ),  
}
```

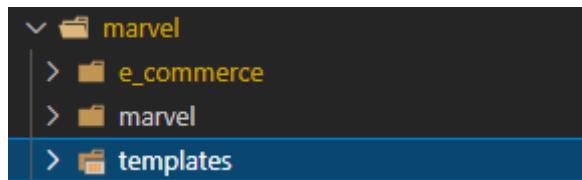
Definimos dentro de `settings.py` el directorio de "templates" para poner nuestro `.html` del frontend de Swagger:

```
TEMPLATES = [  
    {  
        'BACKEND': 'django.template.backends.django.DjangoTemplates',  
        'DIRS': [os.path.join(BASE_DIR,'templates')], # ¡AGREGAMOS ESTA LINEA!  
        'APP_DIRS': True,  
        'OPTIONS': {  
            'context_processors': [  
                'django.template.context_processors.debug',  
                'django.template.context_processors.request',  
                'django.contrib.auth.context_processors.auth',  
                'django.contrib.messages.context_processors.messages',  
            ],  
        },  
    },  
]
```

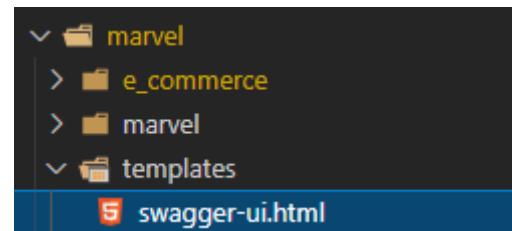
Asegurarse de tener importado el módulo "os" en el archivo settings.py:

```
import os
```

Luego creamos la carpeta "templates" a la altura de nuestro archivo manage.py:



Dentro de "templates" crearemos un nuevo archivo al que llamaremos "swagger-ui.html":



Dentro del archivo swagger-ui.html debemos insertar el siguiente código html:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Inove API</title>
    <meta charset="utf-8"/>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" type="text/css" href="//unpkg.com/swagger-ui-dist@3/swagger-ui.css" />
  </head>
  <body>
    <div id="swagger-ui"></div>
    <script src="//unpkg.com/swagger-ui-dist@3/swagger-ui-bundle.js"></script>
    <script>
      const ui = SwaggerUIBundle({
        url: "{% url schema_url %}",
        dom_id: '#swagger-ui',
        presets: [
          SwaggerUIBundle.presets.apis,
          SwaggerUIBundle.SwaggerUIStandalonePreset
        ],
        layout: "BaseLayout",
        requestInterceptor: (request) => {
          request.headers['X-CSRFToken'] = "{{ csrf_token }}"
          return request;
        }
      })
    </script>
  </body>
</html>
```

Una vez hecho esto, tenemos que declararla en nuestras urls.py, lo más recomendable es hacerlo en la general porque es para una aplicación del sistema.

Importamos TemplateView en urls.py y otro complemento para que funcione con DRF:

```
from django.views.generic import TemplateView
from rest_framework.schemas import get_schema_view
```

Y por último, creamos los path() dentro de nuestras urlpatterns:

```
path('api-docs/', TemplateView.as_view(
    template_name='swagger-ui.html',
    extra_context={'schema_url':'openapi-schema'}
), name='swagger-ui'),
```

Y la otra

```
path('openapi', get_schema_view(
    title="Inove Marvel e-commerce",
    description=description,
    version="1.0.0"
), name='openapi-schema'),
```

La variable "description" la utilizaremos para darle una descripción general a nuestra interfaz de documentación, la cual, admite código HTML, y eso nos abre la puerta a poder la personalización que deseemos, en nuestro caso:

```
description = '''

```

Al ingresar a localhost:8000/api-docs/ podremos ver la interfaz terminada:



Documentación general de APIs de la aplicación e-commerce

Para la autenticación por medio de TOKENS debemos agregar en el header:

- 'Authorization': 'Token 92937874f377a1ea17f7637ee07208622e5cb5e6'

Donde 92937874f377a1ea17f7637ee07208622e5cb5e6 es un ejemplo del Token Key.

Con cada endpoint de la API disponible:

e-commerce	
GET	/e-commerce/comics/get
GET	/e-commerce/comics/get-post
POST	/e-commerce/comics/get-post
GET	/e-commerce/comics/{id}/update
PUT	/e-commerce/comics/{id}/update
PATCH	/e-commerce/comics/{id}/update
GET	/e-commerce/Wish/get
POST	/e-commerce/user/login/
POST	/e-commerce/comics/post
POST	/e-commerce/Wish/post
DELETE	/e-commerce/comics/{id}/delete

y los esquemas de entrada de cada uno:

```

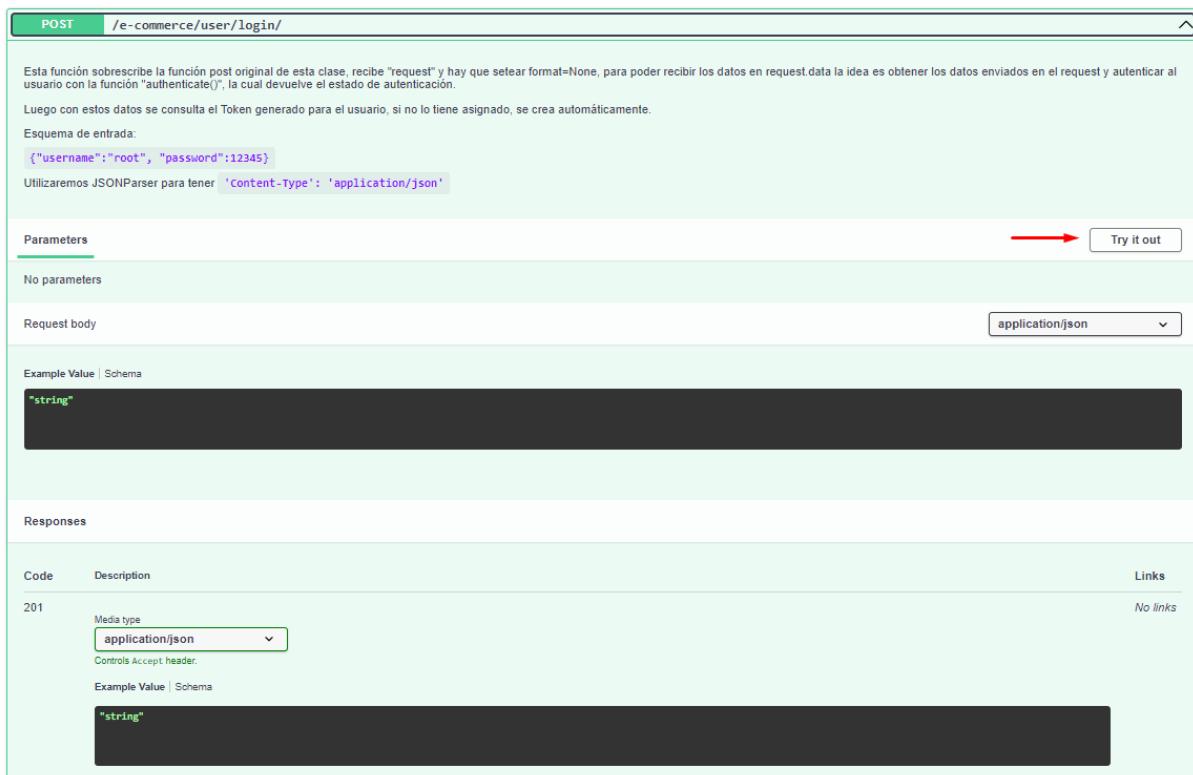
Schemas

Comic ▼ {
    marvel_id      integer
    title          string
    description    string
    price          number
    stock_qty      integer
    picture        string($uri)
    favorite       boolean
}
Comic
{
    marvel_id: 2147483647
    title: "string"
    description: "string"
    price: 0
    stock_qty: 2147483647
    picture: "string($uri)"
    favorite: false
}

WishList ▼ {
    id              integer
    user_id*        integer
    comic_id*       integer
    favorite        boolean
    cart            boolean
    wished_qty     integer
    buied_qty       integer
}
WishList
{
    id: 0
    user_id*: 0
    comic_id*: 0
    favorite: false
    cart: false
    wished_qty: 2147483647
    buied_qty: 0
}

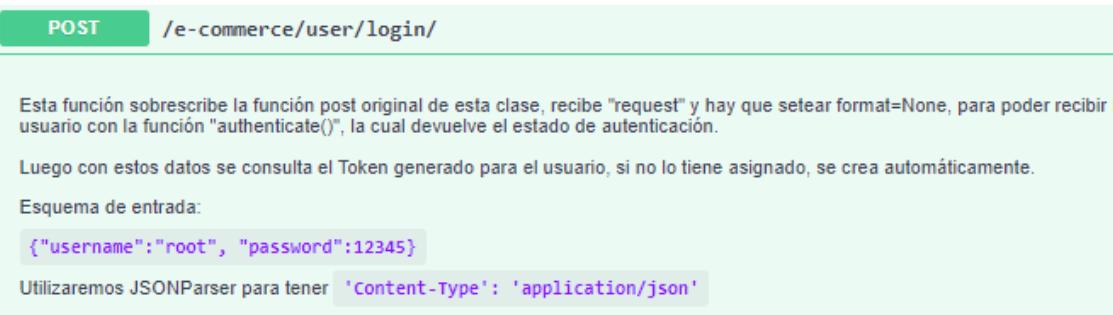
```

Cada endpoint es un botón desplegable que nos permite interactuar con la API:



The screenshot shows a POST request to the endpoint `/e-commerce/user/login/`. The request body contains the JSON object `{"username": "root", "password": "12345"}`. The Content-Type is set to `application/json`. The response code 201 indicates success, with the media type set to `application/json`. A red arrow points to the `Try it out` button.

En el caso de las APIs personalizadas, el input se mostrará como un campo en donde dice "string" pero admite un JSON escrito a mano, el cual enviaremos, si presionamos "try it out" y completamos el campo con los datos que dejamos de ejemplo en la descripción de la API (siempre que los datos sean reales en la base de datos claro):



The screenshot shows a POST request to the endpoint `/e-commerce/user/login/`. The request body contains the JSON object `{"username": "root", "password": "12345"}`. The Content-Type is set to `application/json`. The response code 201 indicates success, with the media type set to `application/json`. A red arrow points to the `Try it out` button.

Descargar un repositorio de GitHub

Para poder realizar las actividades de aquí en adelante debemos tener instalado y configurado nuestro GitHub. Todos los ejemplos prácticos estarán subidos al repositorio GitHub de **InoveAlumnos**, para aprender como descargar estos ejemplos desde el repositorio referirse al "Instructivo de GitHub: Descargar un repositorio" disponible entre los archivos del campus. De no encontrarse allí, por favor, tenga a bien comunicarse con alumnos@inove.com.ar para su solicitud.

Debemos descargar el repositorio que contiene los ejemplos de clase de ésta unidad:

<https://github.com/InoveAlumnos/django-restframework2>

Links de interés

-  [Página oficial](#)
-  [Tutorial de instalación de DRF](#)
-  [Documentación en DRF de PrimaryKeyRelatedField](#)
-  [APIView](#)
-  [Seteo de parceadores \[JSON PARSER\]](#)
-  [Swagger API documentation site](#)