Proyecto API con Django Rest Framework

En este curso haremos uso de algunas apps de Django para implementar una API totalmente funcional utilizando autenticación con Token, muy útil para utilizar en todo tipo de clientes, desde web a aplicaciones móviles, ya que se basa sobre una capa de peticiones con respuestas JSON.

Os enseñaré que no es necesario ser todo un experto para desarrollar la plataforma y en muy poco tiempo tendremos un genial sistema para gestionar películas.

En él los usuarios que se registren e identifiquen podrán marcarlas como favoritas, así como ordenar y buscar en los viewsets gracias a las utilidades de Django Rest Framework.

Repositorio: https://github.com/hektorprofe/curso-api-peliculas-django

Creando nuestro proyecto

Partiremos de la base que ya sabéis crear entornos virtuales con **Pipenv** tal como explico en el primer curso de Django, así que trabajaremos en un directorio llamado **proyecto_pelis** :

```
cd proyecto_pelis
pipenv install django
```

Creamos el proyecto:

```
pipenv run django-admin startproject api_pelis
```

Ahora instalamos todos los módulos de Django para este tutorial, a poder ser con estas versiones para no tener contratiempos:

```
pipenv install djangorestframework, markdown, django-filter
```

Activamos la app *rest_framework* en el **settings.py**.

En este punto os recomiendo crear algunos atajos de comandos en el **Pipfile** para agilizar el desarrollo:

```
Pipfile
[scripts]
dev = "python manage.py runserver 127.0.0.1:8844"
make = "python manage.py makemigrations"
migrate = "python manage.py migrate"
```

Para poner en marcha nuestro proyecto utilizaremos el atajo así:

```
pipenv run dev
```

Si todo va bien deberíamos tener Django en marcha: http://127.0.0.1:8844/.

A continuación crearemos la app que manejará la API:

```
pipenv run python manage.py startapp api
```

La activaremos en el **settings.py** y haremos la migración inicial:

```
pipenv run migrate
```

Finalmente crearemos nuestro usuario administrador:

```
pipenv run python manage.py createsuperuser
```

Modelo de película y serializador

Creamos nuestro modelo *Pelicula* para la API:

```
api/models.py
from django.db import models

class Pelicula(models.Model):
  titulo = models.CharField(max_length=150)
  estreno = models.IntegerField(default=2000)
  imagen = models.URLField(help_text="De imdb mismo")
  resumen = models.TextField(help_text="Descripción corta")

  class Meta:
    ordering = ['titulo']
```

Migramos los cambios:

```
pipenv run make
pipenv run migrate
```

Ahora vamos a configurar un serializador, éste definirá el contenido de las películas tal como las devolverá la API:

```
api/serializers.py
from .models import Pelicula
from rest_framework import serializers

class PeliculaSerializer(serializers.ModelSerializer):
  class Meta:
    model = Pelicula
    # fields = ['id', 'titulo', 'imagen', 'estreno', 'resumen']
```

```
    fields = '__all__'
```

Programando la viewset y las urls

Tenemos el modelo y el serializador, ya sólo nos falta programar el viewset de DRF:

```
api/views.py
from .models import Pelicula
from .serializers import PeliculaSerializer
from rest_framework import viewsets

class PeliculaViewSet(viewsets.ModelViewSet):
  queryset = Pelicula.objects.all()
  serializer_class = PeliculaSerializer
```

Y ahora añadimos a las urls la ruta de la viewset en la API:

```
api_pelis/urls.py
from django.contrib import admin
from django.urls import path, include

from api import views
from rest_framework import routers

router = routers.DefaultRouter()

# En el router vamos añadiendo los endpoints a los viewsets
router.register('peliculas', views.PeliculaViewSet)

urlpatterns = [
  path('api/v1/', include(router.urls)),
  path('admin/', admin.site.urls),
]
```

Ahora podemos navegar a la API para manejar las películas.

Lamentablemente por defecto las viewsets tienen permisos públicos, así que cualquiera podría manejar las películas. Para solucionarlo simplemente añadiremos un permiso por defecto en el **settings.py**:

```
api_pelis/settings.py
REST_FRAMEWORK = {
  'DEFAULT_PERMISSION_CLASSES': [
    'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly',
  ],
}
```

Éste hará nuestras viewsets de sólo lectura incluso para visitantes no autenticados. Sólo los usuarios identificados con suficientes permisos podrán acceder a las acciones de creación, modificación y borrado.

Instalación de las dependencias

La autenticación con token es un sistema ideal para usar en webs asíncronas.

El token es un identificador único de la sesión de un usuario que sirve como credenciales de acceso a la API. Si el cliente envía este token en sus peticiones (que generaremos cuando se registra), ésta buscará si tiene los permisos necesarios para acceder a las acciones protegidas.

Desarrollar esta funcionalidad es tedioso, pero por suerte en DRF tenemos una serie de apps que nos harán la vida mucho más fácil, sólo tenemos que configurarlas adecuadamente y en pocos minutos tendremos un sistema de autenticación básico funcionando.

Esas apps son *django-rest-auth* para la autenticación y *django-allauth* para las cuentas de usuario:

```
pipenv install django-rest-auth
pipenv install django-allauth
```

Configurarlas conlleva añadir algunas apps y variables al settings y paths a las urls.

Primero la app *django.contrib.sites* que maneja los sites requeridos por nuestras dependencias:

```
api_pelis/settings.py
django.contrib.sites,
```

Luego *rest_framework.authtoken* para añadir la autenticación con tokens:

```
rest_framework.authtoken,
```

Dos más de *django-rest-auth*:

```
'rest_auth',
'rest_auth.registration',
```

Y otras tres para *django-allauth*:

```
'allauth',
'allauth.account',
'allauth.socialaccount',
```

Justo debajo de las apps pondremos las siguientes variables:

```
SITE_ID = 1

ACCOUNT_EMAIL_VERIFICATION = 'none'
ACCOUNT_AUTHENTICATION_METHOD = 'email'
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_UNIQUE_EMAIL = True
ACCOUNT_USERNAME_REQUIRED = False
```

En resumen lo que hacemos es añadir la configuración del site actual, simplemente estableciendo un *SITE_ID*. Luego toda la parte que manejará la cuenta, que configuraremos sin verificación de email porque para esta pequeña API nos lo necesitamos, así como registro con email en lugar de usuario, algo que Django no permite hacer por defecto y la verdad es que es un puntazo.

Ahora añadiremos el backend de *allauth* a la configuración de Django, sin ello no nos funcionará nada.

```
AUTHENTICATION_BACKENDS = (
   "django.contrib.auth.backends.ModelBackend",
   "allauth.account.auth_backends.AuthenticationBackend"
)
```

Respecto a las URL añadiremos la parte de la autenticación y registro con *rest_auth*:

```
api_pelis/urls.py
path('api/v1/auth/',
  include('rest_auth.urls')),
path('api/v1/auth/registration/',
  include('rest_auth.registration.urls')),
```

Finalmente migramos de nuevo:

```
pipenv run migrate
```

Si lo hemos hecho todo correctamente, sin mucho problema podremos acceder a la url y registrar un usuario de prueba usando la interfaz web de DRF, de manera que justo después del registro veamos el token generado.

Por cierto, no es necesario poner un Username, sólo el Email y dos contraseñas iguales, eso sí, deben ser bastante seguras.

De la misma forma que registramos un usuario podemos conseguir el token haciendo login en la respectiva URL.

Interactuando usando cURL

La API de Django nos proporciona la interfaz web, pero ¿cómo crearíamos un usuario o haríamos el login desde un cliente?

Para hacer una prueba vamos a usar cURL, una biblioteca que permite hacer peticiones en un montón de protocolos. Normalmente viene instalada en los sistemas operativos, así que sólo tenemos que abrir la terminal para empezar a probar.

Tanto el registro como el login manejan peticiones con métodos POST, podemos crearlas fácilmente.

Para registrar un usuario lo haremos así (es muy importante usar doble comillas al pasar los argumentos):

```
curl -X POST http://127.0.0.1:8844/api/v1/auth/registration/
-d "password1=TEST1234A&password2=TEST1234A&email=test2@test2.com"
```

Y para hacer login y conseguir el token:

```
curl -X POST http://127.0.0.1:8844/api/v1/auth/login/
-d "password=TEST1234A&email=test2@test2.com"
```

La idea trabajando con clientes es crear estas peticiones y almacenar el token en el localStorage del navegador para más adelante pasarlos en las cabeceras de las peticiones que requieran autenticación.

Modelo de película favorita

El sistema de películas favoritas constará de dos vistas: una para marcar o desmarcar una película como favorita y otra que devolverá todas las películas favoritas del usuario.

Obviamente ambas vistas serán de acceso protegido y requerirán pasar el token de autenticación en las cabeceras.

Sin embargo antes de ponernos con las vistas tenemos que crear el nuevo modelo que relacione las películas con los usuarios en forma de favorito:

```
api/models.py
from django.contrib.auth.models import User

class PeliculaFavorita(models.Model):
  pelicula = models.ForeignKey(Pelicula, on_delete=models.CASCADE)
  usuario = models.ForeignKey(User, on_delete=models.CASCADE)
```

Si esta relación entre una película y un usuario existe podremos entender que el usuario la tiene como favorita, y si la desmarca simplemente la borraremos de la base de datos.

Ahora vamos a añadir el serializador del modelo que utilizaremos luego:

```
api/serializers.py
from .models import Pelicula, PeliculaFavorita

class PeliculaFavoritaSerializer(serializers.ModelSerializer):

  pelicula = PeliculaSerializer()

  class Meta:
    model = PeliculaFavorita
    fields = ['pelicula']
```

El punto es que hacemos uso del otro serializador Pelicula dentro de PeliculaFavorita para conseguir que la API devuelva instancias anidadas, ya veréis como es muy útil.

Por ahora lo dejamos así, vamos a migrar antes de continuar con la siguiente lección:

```
pipenv run make
pipenv run migrate
```

Vista de película favorita

Nota: Las peticiones para crear recursos sin un identificador previo son de tipo POST, mientras que las de tipo PUT se usan para crear/reemplazar las que ya existen a partir de un identificador. En otras palabras, no es no se pueda usar PUT, pero al no tener un identificador explícito es mejor utilizar POST tal como lo encontraréis editado en estos apuntes. Si queréis más información echad un vistazo a este enlace.

La vista para manejar una película favorita se basa en definir un método de petición, que en nuestro caso será de tipo PUT.

En él detectaremos una id de película que pasaremos como parámetro para recuperar la película que queremos marcar como favorita.

Una vez la tengamos recuperada crearemos la relación PeliculaFavorita, y si ya existe la borraremos haciendo lo contrario:

```
api/views.py
from .models import Pelicula, PeliculaFavorita
from .serializers import PeliculaSerializer,
PeliculaFavoritaSerializer
```

```
from django.shortcuts import get_object_or_404

from rest_framework import viewsets, views
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response


class MarcarPeliculaFavorita(views.APIView):
  authentication_classes = [TokenAuthentication]
  permission_classes = [IsAuthenticated]

  # POST -> Se usa para crear un recurso sin un identificador
  # PUT -> Se usa para crear/reemplazar un recurso con un
identificador

  def post(self, request):

    pelicula = get_object_or_404(
      Pelicula, id=self.request.data.get('id', 0)
    )

    favorita, created = PeliculaFavorita.objects.get_or_create(
      pelicula=pelicula, usuario=request.user
    )

    # Por defecto suponemos que se crea bien
    content = {
      'id': pelicula.id,
      'favorita': True
    }

    # Si no se ha creado es que ya existe, entonces borramos el
favorito
    if not created:
      favorita.delete()
      content['favorita'] = False

    return Response(content)
```

Haciendo uso de una *APIView* genérica configuraremos el sistema de autenticación y de permisos de la vista para que sea accesible sólo a usuarios autenticados vía token. Ya en la respuesta devolveremos una estructura JSON usando la clase *Response* de DRF.

El nuevo path para acceder a la view quedará así:

```
api_pelis/urls.py
path('api/v1/favorita/', views.MarcarPeliculaFavorita.as_view()),
```

Una vez configurada, si accedemos a la URL veremos como indica explícitamente la autenticación con token para tener acceso a ella.

Antes de probar si podemos añadir películas favoritas a través de cURL vamos a crear algunas películas. Normalmente añadiríamos una configuración parael adminitrador, pero ya que tenemos la interfaz de DRF nos la podemos ahorrar. Sólo tenemos que acceder al administrador con nuestro super usuario y luego crear las películas desde la interfaz de DRF.

Para crear una petición PUT con cURL hay que estructurarla de la siguiente forma, pasando el token del usuario que marcará una película en las cabeceras:

```
curl -X POST http://127.0.0.1:8844/api/v1/favorita/
-H "Authorization: Token d0e501a9164b2eeef7f87b62581fb391385fd262"
-d "id=1"
```

Se supone que si todo es correcto nos devolverá el estado de la película:

```
{"id":1,"favorita":true}
```

¿Cómo podemos saber si realmente se creó? La forma fácil sería añadir a la base de datos. Usando DB Browser for SQLite es muy fácil abrir el fichero de la base de datos y analizar los datos de las tablas.

Como hemos comprobado el registro existe, si volvemos a ejecutar la petición debería borrarla:

```
{"id":1,"favorita":false}
```

Vista de películas favoritas

Nuestra siguiente tarea es una vista que devuelva todas las películas favoritas del usuario. Por suerte gracias al serializador que creamos antes es muy fácil:

```
api/views.py
class ListarPeliculasFavoritas(views.APIView):
  authentication_classes = [TokenAuthentication]
  permission_classes = [IsAuthenticated]

  # GET -> Se usa para hacer lecturas

  def get(self, request):

    peliculas_favoritas = PeliculaFavorita.objects.filter(
      usuario=request.user)
    serializer = PeliculaFavoritaSerializer(
```

```
        peliculas_favoritas, many=True)

    return Response(serializer.data)
```

Y el path:

```
api_pelis/urls.py
path('api/v1/favoritas/', views.ListarPeliculasFavoritas.as_view()),
```

Se trata de una simple acción de tipo GET donde devolvemos la lista de pelis favoritas serializadas. Para probar si funciona haremos lo propio con cURL:

```
curl -X GET http://127.0.0.1:8844/api/v1/favoritas/
-H "Authorization: Token d0e501a9164b2eeef7f87b62581fb391385fd262"
```

Esto debería devolvernos la lista con la información de todas las películas favoritas, gracias a lo que os comenté de los modelos anidados:

```
[
   {
      "pelicula":{
         "titulo":"El Padrino",
         "imagen":"https://m.media-amazon.com/images/...",
         "estreno":1972
      }
   }
]
```

Como veis no ha sido para nada complejo.

Top de películas favoritas y filtros

Esta podría bien ser la parte más compleja del proyecto, pues tenemos que encontrar una forma de contar el número de favoritos de cada película y devolver la lista de películas ordenadas a partir de ese campo.

Por suerte me sé un truquillo que nos hará la vida mil veces más fácil.

Lo que vamos a hacer es crear un nuevo campo en la película que almacene el número de favoritos. Este campo lo actualizaremos automáticamente al crearse o borrarse una instancia de PeliculaFavorita.

```
api/models.py
favoritos = models.IntegerField(default=0)
```

Luego abajo del todo crearemos la señal que hará toda la magia, actualizando el contador a partir de una consulta inversa de esas que tanto me gustan:

```
api/models.py
from django.db.models.signals import post_save, post_delete

def update_favoritos(sender, instance, **kwargs):
  count = instance.pelicula.peliculafavorita_set.all().count()
  instance.pelicula.favoritos = count
  instance.pelicula.save()

# en el post delete se pasa la copia de la instance que ya no existe
post_save.connect(update_favoritos, sender=PeliculaFavorita)
post_delete.connect(update_favoritos, sender=PeliculaFavorita)
```

Migramos los cambios:

```
pipenv run make
pipenv run migrate
```

Como utilizamos el campo **all** en el serializador se supone que ya nos devolverá automáticamente este nuevo campo.

Y ya lo tenemos, el detalle maestro lo pondremos añadiendo un par de filtros de ordenamiento y búsqueda al ViewSet de películas que DRF maneja automáticamente. Así permitiremos ordenar las películas por número de favoritos y realizar búsquedas a partir del título:

```
api/views.py
from rest_framework import viewsets, views, filters

class PeliculaViewSet(viewsets.ModelViewSet):
  filter_backends = [filters.SearchFilter, filters.OrderingFilter]
  search_fields = ['titulo']
  ordering_fields = ['favoritos']
```

Ahora desde la interfaz web de DRF podemos filtrar y ordenar las películas.