Proyecto API con Django Rest Framework

A mediados de 2020, en un intento colaborativo por lanzar un curso sobre Django Rest Framework y React, realicé la API que os voy a enseñar a crear en esta sección.

El curso no llegó a buen puerto, pero como me niego a dejar en el olvido todo el material que preparé durante un mes, voy a compartir con vosotros mi parte del curso sobre cómo crear una API para un portal de películas con Django Rest Framework.

Repositorio

Puedes consultar el código fuente en https://github.com/hektorprofe/mispelis-backend.

Espero que os sirva.

Sistema de autenticación y registro en DRF

En esta unidad vamos a implementar todo el sistema de autenticación de usuarios de nuestra API, el cuál controlará diferentes aspectos como:

- El registro
- El login y logout
- La recuperación de contraseña

Toda nuestra API gira alrededor de este sistema, por eso nos tomaremos el tiempo necesario para explicar cuidadosamente cada paso de su desarrollo. Podéis estar seguros de que lo aprendido aquí os servirá en prácticamente todas vuestras futuras APIs creadas con DRF.

Entorno y proyecto

Vamos a partir de una carpeta llamada **mispelis/** que vamos a abrir en VSC , y desde ahí abriremos una terminal **CMD**:

```
# Instalamos Pipenv
pip install pipenv
```

A continuación:

```
# Creamos un entorno instalando django
pipenv install django
```

Creamos el proyecto:

```
# Le llamaremos "server"
pipenv run django-admin startproject server
```

Ahora configuraremos el entorno virtual en VSC abriendo el fichero **manage.py**.

Al abrirlo nos aparecerá en la parte inferior de VSC el intérprete activo, hacemos clic, buscamos uno llamado **server** y lo activamos.

Veremos como a partir de ahora nos aparece **server-XXXXXX** abajo, eso significa que todos los ficheros Python del proyecto se están ejecutando con el intérprete de nuestro entorno virtual.

Inmediatamente después de configurar el entorno, nos pedirá instalar **pylint** en las dependencias, lo hacemos, ya que nos señalará posibles fallos en el código.

Bien, vamos a comprobar si funciona nuestro proyecto:

```
cd server
pipenv run python manage.py runserver
```

Si todo funciona podremos acceder a `http://127.0.0.1:8000/`. Podemos cambiar el idioma del proyecto:

mispelis/server/settings.py

```
LANGUAGE_CODE = 'es'
```

Justo después de guardar el primer fichero python del proyecto VSC nos pedirá instalar la extensión **autopep8**, la instalamos. Este paquete nos formateará el código automáticamente cumpliendo las pautas definidas en el pep8.

Si todo ha ido bien deberíamos tener **django** ya funcionando en español.

Por último en esta lección vamos a crear unos scripts en nuestro entorno para manejar más fácilmente **django**.

mispelis/Pipfile

```
[scripts]
server          = "python manage.py runserver"
migrate         = "python manage.py migrate"
startapp        = "python manage.py startapp"
makemigrations  = "python manage.py makemigrations"
createsuperuser = "python manage.py createsuperuser"
```

Cerramos el servicio y testeamos:

```
pipenv run server
```

Con esto es suficiente para empezar a programar nuestra API.

App de autenticación

Vamos a empezar nuestro desarrollo con una **app** que maneje la autenticación, la parte de la API que manejará las funcionalidades de:

- Registrar una cuenta
- Iniciar y cerrar sesión
- Restaurar contraseña

Empezamos por aquí porque la API que vamos a desarrollar es privada, accesible sólo a usuarios registrados e identificados. Nos os preocupéis por esto, ya que eventualmente también veremos cómo librerar una vista y hacerla accesible sin autenticación.

Dicho lo cuál, vamos crear la app de autenticación:

```
pipenv run startapp authentication
```

Podéis ponerle cualquier nombre menos **auth** a secas, ya que ese nombre es el de una app interna **django.contrib.auth** y no debemos sobreescribirlo.

La activamos debajo de las **apps** por defecto de Django:

server/settings.py
```
INSTALLED_APPS = [

    # Django internal apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Django custom apps
    'authentication',
]
```

Antes de ponernos con las vistas vamos a dejar instalado y configurado **Django Rest Framework**, la app que nos permite transformar Django en un servidor de APIs:

```
cd server
pipenv install djangorestframework
```

La activamos:

server/settings.py
```
INSTALLED_APPS = [

    # Django internal apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Django external apps
    'rest_framework',
```

```
    # Django custom apps
    'authentication',
]
```

Muy bien, en la siguiente lección vamos a crear el modelo para los usuarios de nuestra aplicación.

Custom User

El modelo de usuario que trae Django es algo limitado y nosotros necesitamos añadir algunos campos nuevos. Una forma de hacerlo es crear un modelo **Profile** enlazado a cada usuario con los campos que queremos, sin embargo en este curso vamos a hacerlo de una forma más profesional: **personalizar el usuario base**.

IMPORTANTE: Antes de continuar es clave no haber creado ningún usuario en la base de datos. Si lo habéis hecho, borrad la base de datos **db.sqlite3** y todos los ficheros del directorio **migrations** excepto los llamados `__init__.py`.

Para extender el modelo de usuario, vamos a crear un **CustomUser** en nuestra app. La diferencia más importante respecto a un usuario "clásico" es que, si bien Django maneja como campo identificador el **username**, nosotros lo identificaremos con el **email**.

authentication/models.py
```
from django.contrib.auth.models import AbstractUser
from django.db import models

class CustomUser(AbstractUser):
    email = models.EmailField(
        max_length=150, unique=True)
```

Según la documentación para establecer el campo identificador único del **User** tenemos que definir lo siguiente:

authentication/models.py
```
    USERNAME_FIELD = 'email'  # new
```

Con estos cambios los usuarios podrán iniciar sesión con el correo en lugar de usar su username.

En este punto repasemos los campos obligatorios del formulario de registro:

- Email: que debe ser único y se utilizará como identificador para acceder.
- Username: que también debe ser único y se mostrará en el perfil.
- Password: para la contraseña del usuario.

Según la documentación, para obligar al usuario a introducir un campo debemos indicarlo en una lista:

authentication/models.py
```
    REQUIRED_FIELDS = ['username', 'password']  # new
```

Fijaros que hacemos referencia a **username** y **password** pero nunca los definimos. Eso es porque se heredan del **AbstractUser**. De hecho el propio **email** también se hereda, pero como por defecto no es obligatorio ni único, tenemos que cambiar su configuración tal como hemos hecho.

Ahora vamos a decirle a Django que utilice el **CustomUser** en lugar de su modelo genérico:

server/settings.py
```
# Custom user model
AUTH_USER_MODEL = "authentication.CustomUser"
```

Hacemos la migración inicial junto con nuestra app:

```
cd server
pipenv run makemigrations
pipenv run migrate
```

Vamos a crear un usuario admin para acceder a nuestra administración. Si todo va bien debería pedirnos el email como identificador, a parte de un username:

```
cd server
pipenv run createsuperuser

> Email: admin@admin.com
> Nombre de usuario: admin
> Contraseña: 1234
> Superuser created successfully.
```

Perfecto, si ponemos en marcha el servidor deberíamos ser capaces de acceder al panel de administración `http://127.0.0.1:8000/admin/`

Al hacerlo notaremos no podemos gestionar usuarios automáticamente. Algo entendible al haber creado nuestro propio modelo.

Para tener de el admin hay que configurarlo manualmente como un modelo cualquiera, la ventaja es que podemos heredar el del usuario por defecto.

No entraremos en detalle porque asumo que la mayoría tenéis conocimientos básicos sobre el panel de administrador. Si no es el caso os recomiendo mi otro curso de introducción a Django.

authentication/admin.py
```
from django.contrib import admin
from django.contrib.auth import get_user_model
from django.contrib.auth.admin import UserAdmin


@admin.register(get_user_model())
class CustomUserAdmin(UserAdmin):
    pass
```

En cualquier caso ya deberíamos tener acceso a nuestro modelo para crear, editar y borar usuarios desde el administrador. Y si queréis incluso podemos desactivar los grupos porque no los vamos a utilizar:

authentication/admin.py

```
from django.contrib.auth.models import Group  # new

admin.site.unregister(Group)  # new
```

Ya estamos listos para empezar con las vistas de la API, pero antes tenemos que tomar una decisión muy importante.

Sistemas de autenticación

Las APIs REST como la que vamos a crear permiten implementar diferentes protocolos de autenticación. Vamos a repasar brevemente los más famosos:

Básica

Es el método más sencillo pero inseguro, ya que se basa en enviar el usuario y su contraseña codificadas en **Base64** en las cabeceras de cada petición. Tened en cuenta que la codificiación en Base64 es una formalidad y es tan sencillo descodificar una cadena en este formato como pasar el valor a una función de cualquier lenguaje de programación.

```
Authorization: Basic base64(username:password)
```

API Keys

Otro sistema bastante utilizado es enviar una **key** facilitada por la API que sustituye las credenciales de un usuario. Muchas APIs públicas ofrecen este sistema porque es fácil añadirle un límite de peticiones diarias, pero en la práctica es sustituir la cadena de autenticación por la clave que se provee a cliente.

```
Authorization: Apikey <key>
```

Bearer

También conocida como *Token Authentication* se basa en proveer al cliente de un "código" después de identificarse por primera vez. En lugar de enviar todo el rato el usuario y su contraseña, el cliente envía ese código generado por el backend en la cabecera de las peticiones. Es igual de inseguro que la autenticación básica, pero como mínimo no se mandan las credenciales en cada petición.

```
Authorization: Bearer <token>
```

JSON Web Tokens

Los JWT son un estándar abierto de la industria para representar peticiones de forma "segura" entre dos partes. Este sistema añade una capa extra de seguridad en los tokens, con mecanismos de decodificación, verificación y renovación de los mismos.

```
Authorization: Bearer <JWT token>
```

OAuth2

Este sistema de autenticación permite compartir información entre sitios sin compartir las credenciales del usuario. En lugar de generar el token en nuestro backend se delega este proceso una plataforma de terceros, de manera que no necesitamos almacenar el usuario y la contraseña en la base de datos. Oauth2 es muy cómodo, pero si tenemos interés en almacenar información del usuario en nuestra base de datos igualmente necesitaremos implementar un sistema de usuarios y pedirle la información a esas plataformas de terceros. Casi nunca se utiliza como sistema único de autenticación, sino como apoyo de nuestro propio sistema.

`Authorization: Bearer <Oauth2 token>`

El mejor sistema de autenticación para API

Con esto hemos cubierto prácticamente todos los sistemas de autenticación. ¿Cuál vamos a utilizar? Pues... ninguno de ellos.

Veréis, la gracia de las APIs es que permiten separar la lógica del servidor y la del cliente. Sin embargo esto tiene un problema inherente y es que los clientes deben almacenar las credenciales de acceso en la memoria para poder enviarlas en las peticiones.

Que la información se encuentre almacenada en el cliente implica un defecto de facto en un cliente web y es que esas credenciales SIEMPRE son accesibles a través de JavaScript y por tanto son vulnerables a ser accedidas utilizando ataques XSS (Cross-site Scripting), os dejo un enlace.

Por tanto el problema radica en la propia naturaleza de JavaScript... ¿Cómo lo solucionamos? Pues haciendo inaccesibles las credenciales en el cliente.

Espera espera Héctor... ¿Eso se puede hacer? ¿Cómo va a enviar las credenciales el cliente si no tiene acceso a ellas? La respuesta está en las **cookies**.

Ya sé lo que estáis pensando algunos... las cookies también son accesibles desde Javascript y por tanto vulnerables a ataques XSS, así que no vamos a solucionar nada. Sin embargo, y por suerte para nosotros, existe una cláusula en las cookies llamada **HttpOnly** que Django activa automáticamente en las sesiones. Esta opción hace que las cookies sean inaccesibles desde JavaScript y que sea el propio navegador quien las gestiona.

Con todo esto lograremos un sistema muy robusto, seguro y fácil de implementar.

Solo debemos indicar en las peticiones de JavaScript que deseamos enviar las credenciales de la cookie de sesión. Por ejemplo usando `axios` podemos preconfigurarlo así:

```
axios.defaults.baseURL = "http://localhost:8000/api";
axios.defaults.withCredentials = true;
```

O en cada petición enviar el campo de esta forma:

```
axios.get('http://localhost:8000/api', {withCredentials: true});
```

¡Gracias Django!

Login y logout

Vamos a empezar creando unas vistas básicas de **login** y **logout** usando una **APIView** básica de DRF. La forma de implementar la lógica es exactamente igual que con Django clásico, os dejo el enlace a la documentación oficial por si queréis profundizar:

authentication/views.py

```
from django.contrib.auth import authenticate, login, logout
from rest_framework import status
from rest_framework.response import Response
from rest_framework.views import APIView


class LoginView(APIView):
    def post(self, request):
        # Recuperamos las credenciales y autenticamos al usuario
        email = request.data.get('email', None)
        password = request.data.get('password', None)
        user = authenticate(email=email, password=password)

        # Si es correcto añadimos a la request la información de
sesión
        if user:
            login(request, user)
            return Response(
                status=status.HTTP_200_OK)

        # Si no es correcto devolvemos un error en la petición
        return Response(
            status=status.HTTP_404_NOT_FOUND)


class LogoutView(APIView):
    def post(self, request):
        # Borramos de la request la información de sesión
        logout(request)

        # Devolvemos la respuesta al cliente
        return Response(status=status.HTTP_200_OK)
```

Configuramos las dos URL en la app:

authentication/urls.py

```
from django.urls import path, include
from .views import LoginView, LogoutView

urlpatterns = [
    # Auth views
    path('auth/login/',
         LoginView.as_view(), name='auth_login'),

    path('auth/logout/',
```

```
        LogoutView.as_view(), name='auth_logout'),
]
```

Y configuramos las URL de la app en el proyecto:

server/urls.py
```
from django.contrib import admin
from django.urls import path, include
from rest_framework import routers

# Api router
router = routers.DefaultRouter()

urlpatterns = [
    # Admin routes
    path('admin/', admin.site.urls),

    # Api routes
    path('api/', include('authentication.urls')),
    path('api/', include(router.urls)),
]
```

Con esta estructura tenemos dos endpoints:

- `/api/auth/login/`
- `/api/auth/logout/`

En la siguiente lección probaremos si funcionan correctamente.

Probando la autenticación

Ha llegado la hora de probar la API, la forma más fácil es desde la interfaz que nos provee DRF.

Acceded a la URL de **login** `http://localhost:8000/api/auth/login/` y escribid las credenciales en crudo como si fuera un objeto JSON:
```
{ "email": "admin@admin.com", "password": "1234" }
```

Si todo funciona correctamente al enviar el formulario veréis la respuesta de la API y saldrá vuestro email arriba a la derecha indicando que efectivamente estamos identificados:

```
{ "login": "success" }
```
Para probar el **logout**, estando identificados, accedemos a la URL pertinente `http://localhost:8000/api/auth/logout/` y al enviar el formulario vacío debería hacernos lo propio:
```
{ "logout": "success" }
```

Si nos logeamos ahora nos devolverá la información que hemos establecido en el serializador:

```
{
  "email": "admin@admin.com",
  "username": "admin",
  "password": "pbkdf2_sha...."
}
```

Como el campo password no nos interesa serializado vamos a establecer una clásula de sólo escritura, para que Django sólo lo tenga en cuenta al crear o modificar, pero nunca al hacer una lectura:

```
password = serializers.CharField(
    min_length=8, write_only=True)
```

Serializando el usuario

Uno de los requisitos del frontend es que justo después de identificarnos la API debe enviar información básica del usuario para utilizarla en la aplicación, como por ejemplo el nombre, el email o más adelante el avatar.

Cuando nos autenticamos conseguimos un objeto **user** con toda esa información, pero no podemos enviarlo al cliente y ya está, necesitamos transformarlo a un objeto JSON. Ese proceso de transformar el objeto de un formato a otro, Python a Javascript en nuestro caso, se conoce como seralización.

DRF permite crear serializadores de modelos para automatizar esta tarea, así que vamos a crear nuestro propio serializador de usuarios:

authentication/serializers.py

```
from rest_framework import serializers
from django.contrib.auth import get_user_model


class UserSerializer(serializers.ModelSerializer):
    email = serializers.EmailField(
        required=True)
    username = serializers.CharField(
        required=True)
    password = serializers.CharField(
        min_length=8)

    class Meta:
        model = get_user_model()
        fields = ('email', 'username', 'password')
```

Este serializador básico, a parte de controlar los campos que queremos serializar de Python a JSON, también nos servirá más adelante para configurar métodos como la creación de usuarios durante el registro y la validación personalizada de campos.

Sea como sea vamos a serializar el objeto **user** y a enviarlo como respuesta de la petición de login:

authentication/views.py

```
from .serializers import UserSerializer

# ...

return Response(
    UserSerializer(user).data,
    status=status.HTTP_200_OK)
```

Registro de usuarios

En esta lección vamos a por la vista de registro, donde a partir de un email, username y password crearemos usuarios en la base de datos.

DRF tiene una vista llamada **CreateAPIView** que automatiza la tarea de crear instancias a partir de un serializador, vamos a usarla para facilitarnos la vida:

authentication/views.py
```
from rest_framework import generics, status

# ...

class SignupView(generics.CreateAPIView):
    serializer_class = UserSerializer
```

Añadimos la vista a la URL y a probar si funciona:

authentication/urls.py
```
from django.urls import path, include
from .views import LoginView, LogoutView, SignupView

urlpatterns = [
    # ...
    path('auth/signup/',
         SignupView.as_view(), name='auth_signup'),
]
```

Ahora probamos el formulario de registro en el endpoint `http://localhost:8000/api/auth/signup/` y...

```
Email: test@test.com
Username: test
Password: 12345678
```

¡Parece que funcionó! Pero no cantemos victoria, vamos a revisar desde el panel de administración si está todo correcto.

Al entrar al panel de administración vemos que efectivamente el usuario existe, sin embargo al acceder más a fondo podemos comprobar un aviso preocupaunte:

```
Formato de clave incorrecto o algoritmo de hash desconocido.
```

¿Sabéis que ha pasado? Que las contraseñas de usuario en Django se tienen que guardar encriptadas y nosotros la hemos guardado en crudo.

Para codificar la contraseña se me ha ocurrido una forma muy sencilla. Podemos añadir un validador al campo password y utilizar el método **make_password** de Django que lo codifica él solito:

authentication/serializers.py
```
from django.contrib.auth.hashers import make_password

# ...

def validate_password(self, value):
    return make_password(value)
```

Voy a **borrar el usuario test** y a probar si esta vez funciona bien:

```
Email: test@test.com
Username: test
Password: 12345678
```

Y...

```
{
  "email": "test@test.com",
  "username": "test"
}
```

Si consultamos el administrador ya no aparece el error de codificación y podemos iniciar sesión sin problema:

```
{ "email": "test@test.com", "password": "12345678" }
```

¡Listo!

Recuperar contraseña

Un buen sistema de autenticación debe proveer una función de recuperación de contraseña para usuarios despistados.

Una funcionalidad así requiere bastante planificación, crear varias vistas, confirmaciones por email, etc. Sin embargo investigando un poco encontré una **app** de Django que nos hará la mayor parte del trabajo, vamos a instalarla y configurarla:

```
cd server
pipenv install django-rest-passwordreset
```

Una vez instalada la añadimos a las **apps** del proyecto:

server/settings.py

```
INSTALLED_APPS = [

    # Django internal apps
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',

    # Django external apps
    'rest_framework',
    'django_rest_passwordreset',  # new

    # Django custom apps
    'authentication',
]
```

Según la documentación de esta app, ahora tenemos que migrar para crear algunos campos en la base de datos:

```
cd server
pipenv run migrate
```

Añadir un endpoint para manejar las vistas de recuperar contraseña:

authentication/urls.py

```
urlpatterns = [
    # ...
    path('auth/reset/',
         include('django_rest_passwordreset.urls',
                 namespace='password_reset')),
]
```

Con esto podemos acceder al endpoint de la API para recuperar una contraseña y probar si funciona `http://localhost:8000/api/auth/reset/`:

```
test@test.com
```

Esta petición nos devolverá:

```
{ "status": "OK" }
```

¿Qué ha ocurrido? ¿Se supone que debería haber enviado mágicamente un correo? Pues no, esta petición sólo genera el token de recuperación en la tabla de la app. Podemos confirmarlo consultando el panel de administrador.

Para completar el ciclo falta un paso, enviar el correo al cliente cuando se crea el token. Según la documentación de la app, se puede lograr configurando una de la siguiente forma:

```
from django.dispatch import receiver
from django_rest_passwordreset.signals import
reset_password_token_created

#...

@receiver(reset_password_token_created)
def password_reset_token_created(sender, instance,
reset_password_token, *args, **kwargs):
    # Aquí deberíamos mandar un correo al cliente...
    print(
        f"\nRecupera la contraseña del correo
'{reset_password_token.user.email}' usando el token
'{reset_password_token.key}' desde la API
http://localhost:8000/api/auth/reset/confirm/.")
```

Con esto es suficiente para probar la funcionalidad, probad de nuevo a recuperar la contraseña `http://localhost:8000/api/auth/reset/` y veréis el mensaje por la terminal del servidor:

```
Recupera la contraseña del correo test@test.com desde la API
`http://localhost:8000/api/auth/reset/confirm/` usando el token
73538969130d6e9d4a6299a343d512af15b8.
```

A través del enlace a la API podemos acceder al formulario, escribir el token con la nueva contraseña y debería hacernos el cambio. Eso sí, tened en cuenta que el validador de esta app es más exigente y si la contraseña no es bastante segura nos la tumbará devolviendo varios errores:

```
Contraseña: Test1234
```

```
Token: 73538969130d6e9d4a6299a343d512af15b8
```

Con esto tenemos la funcionalidad cubierta, sólo faltaría configurar un cliente de correo en Django y pulir la señal para enviar emails en lugar de mostrar ese **print** en la terminal.

Peticiones CORS

Al acceder a la API desde un cliente web, tal como la tenemos ahora, se nos responderá con este error:

```
Access to XMLHttpRequest at 'http://localhost:8000/api/auth/login/'
from origin 'http://localhost:3000' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No
'Access-Control-Allow-Origin' header is present on the requested
resource.
```

El error **Access-Control-Allow-Origin** indica que se ha bloqueado la petición por ser de tipo CORS (Cross Origin Resource Sharing). Esto sucede porque Django no permite peticiones desde distinto hosts y esto es algo que afecta tanto al dominio como al puerto. Al correr Django en el 8000 y el cliente React en el 3000, los toma como dos hosts diferentes y salta el error.

Para solucionar este problema tenemos que activar estas peticiones CORS y lo vamos a hacer gracias a una app externa que facilita mucho la configuración:

```
cd server
pipenv install django-cors-headers
```

La activamos:

mispelis/server/settings.py
```
INSTALLED_APPS = [
    # ...

    # Django external apps
    'corsheaders',
    'rest_framework',
    'django_rest_passwordreset',

    # ...
]
```

La documentación oficial explica que tenemos que configurar un middleware con preferencia, el cuál se encargará de procesar las peticiones CORS:

```
MIDDLEWARE = [
    'corsheaders.middleware.CorsMiddleware',
    # ...
]
```

Finalmente hay que añadir los dominios que queremos permitir en las peticiones CORS, en nuestro caso el de la URL del cliente:

```
# Configuración de CORS
```

```
CORS_ORIGIN_WHITELIST = ["http://localhost:3000"]
```

Si probamos de nuevo nos devolverá un error diferent al anterior:

```
Access to XMLHttpRequest at 'http://localhost:8000/api/auth/login/'
from origin 'http://localhost:3000' has been blocked by CORS policy:
The value of the 'Access-Control-Allow-Credentials' header in the
response is '' which must be 'true' when the request's credentials
mode is 'include'. The credentials mode of requests initiated by the
XMLHttpRequest is controlled by the withCredentials attribute.
```

El error ahora se llama **'Access-Control-Allow-Credentials'** y nos pide quye se puedan gestionar las credenciales de la sesión en las cabeceras de las peticiones CORS. Hacerlo es tan fácil como añadir una simple línea:

```
CORS_ALLOW_CREDENTIALS = True
```

Debido a que estamos usando un sistema de autenticación clásico de Django, éste espera que el cliente maneje las credenciales con el atributo **withCredentials** en las peticiones, tal como indicaba la parte final del error:

```
The credentials mode of requests initiated by the XMLHttpRequest is
controlled by the withCredentials attribute.
```

Tampoco debemos olvidar que **Django** implementa un sistema de seguridad contra exploits **CSRF** (Cross-Site Request Forgery), por lo que en las peticiones se espera recibir de vuelta una cookie con el **csrftoken**, pero eso es algo que se configura en el cliente. Si queréis saber más sobre esta vulernabilidad os dejo este enlace.

Sea como sea ya tenemos la configuración lista para la mayor parte del desarrollo.

Sistema de películas y géneros en DRF

En esta unidad crearemos la segunda app del proyecto, encargada de manejar las películas y sus géneros. También veremos cómo proteger las vistas haciéndolas privadas y os facilitaré una base de datos con docenas de películas ya creadas.

App Films

Creamos la app:

```
cd server
pipenv run startapp films
```

La activamos:

server/settings.py
```
INSTALLED_APPS = [
```

```
    # Django custom apps
    'authentication',
    'films',
]
```

Vamos a crear dos modelos, uno para las películas y otro para sus géneros. Cada película podrá tener varios géneros, por lo que usaremos un modelo dentro de otro:

films/models.py

```
import uuid
from django.db import models
from django.utils.text import slugify


class Film(models.Model):

    id = models.UUIDField(  # uuid en lugar de id clásica
autoincremental
        primary_key=True, default=uuid.uuid4, editable=False)
    title = models.CharField(
        max_length=150, verbose_name="Título")
    year = models.PositiveIntegerField(
        default=2000, verbose_name="Año")
    review_short = models.TextField(
        null=True, blank=True, verbose_name="Argumento (corto)")
    review_large = models.TextField(
        null=True, blank=True, verbose_name="Historia (largo)")
    trailer_url = models.URLField(
        max_length=150, null=True, blank=True, verbose_name="URL
youtube")
    genres = models.ManyToManyField(
        'FilmGenre', related_name="film_genres",
verbose_name="Géneros")

    class Meta:
        verbose_name = "Película"
        ordering = ['title']

    def __str__(self):
        return f'{self.title} ({self.year})'


class FilmGenre(models.Model):
    name = models.CharField(
        max_length=50, verbose_name="Nombre", unique=True)
    slug = models.SlugField(
        unique=True)

    class Meta:
        verbose_name = "género"
        ordering = ['name']

    def __str__(self):
        return f'{self.name}'

    def save(self, *args, **kwargs):
        self.slug = slugify(self.name)
        super(FilmGenre, self).save(*args, **kwargs)
```

Damos de alta los modelos en el panel de administrador para gestionarlos:

films/admin.py
```
from django.contrib import admin
from .models import Film, FilmGenre


@admin.register(Film)
class FilmAdmin(admin.ModelAdmin):
    pass


@admin.register(FilmGenre)
class FilmGenreAdmin(admin.ModelAdmin):
    readonly_fields = ["slug"]
```

Hacemos las migraciones y migramos la app:

```
cd server
pipenv run makemigrations films
pipenv run migrate films
```

Con esto ya tenemos los modelos preparados, sin embargo en la siguiente lección nos tomaremos un rato para añadir un par de campos para almacenar unas imágenes en nuestras películas.

Miniaturas y wallpapers

En esta lección vamos a añadir dos imágenes a las películas, una miniatura con la carátula y otra con un fondo de pantalla. Quedarán genial en el cliente web, ya veréis.

Si queremos almacenar ficheros en los modelos tenemos que configurar los ficheros **media**:

server/settings.py
```
import os

# Media files
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')  # path al directorio
local
MEDIA_URL = 'http://localhost:8000/media/'    # url para el desarrollo
```

Según nuestra configuración los ficheros **media** se almacenarán en un directorio con ese mismo nombre, ubicado en la raiz del proyecto de Django. Lo creamos:

```
cd server
mkdir media
```

A continuación tenemos que hacer que el servidor de **Django** sirva los ficheros del directorio **media** o no podremos ver las imágenes:

server/urls.py
```
from django.conf import settings
from django.conf.urls.static import static
```

```
# ...

# Serve static files in development server
if settings.DEBUG:
    urlpatterns += static('/media/',
document_root=settings.MEDIA_ROOT)
```

Esta configuración es sólo para la fase de desarrollo, en producción los ficheros los servirá un servidor especializado como **Nginx** o **Apache**.

Ya estamos listos para añadir nuestros dos campos de imágenes:

films/models.py
```
image_thumbnail = models.ImageField(
    upload_to='films/', null=True, blank=True,
verbose_name="Miniatura")
image_wallpaper = models.ImageField(
    upload_to='films/', null=True, blank=True,
verbose_name="Wallpaper")
```

Sin embargo de esta forma todas las imágenes de las películas se guardarán en el mismo directorio y eso no es muy eficiente.

Una mejor aproximación es crear una carpeta para cada modelo usando algún campo. En nuestro caso podríamos usar el identificador de la película:

Para conseguirlo implementaremos un método que devuelva el path final al campo **upload_to**. El truco está en que se envían implícitamente la instancia y el nombre del fichero para que podamos generar la estructura que deseemos:

films/models.py
```
class Film(models.Model):

    def path_to_film(self, instance, filename):
        return f'films/{instance.id}/{filename}'

    # ...

    image_thumbnail = models.ImageField(
        upload_to=path_to_film, null=True, blank=True,
verbose_name="Miniatura")
    image_wallpaper = models.ImageField(
        upload_to=path_to_film, null=True, blank=True,
verbose_name="Wallpaper")
```

Una vez actualizado el modelo creamos de nuevo las migraciones:

```
cd server
pipenv run makemigrations films
pipenv run migrate films
```

Nos saltará un error:

```
Cannot use ImageField because Pillow is not installed.
```

Django necesita el módulo **Pillow** para manipular imágenes, así que vamos a instalarlo (puede tardar un poco). Luego migramos de nuevo:

```
cd server
pipenv install Pillow
pipenv run makemigrations films
pipenv run migrate films
pipenv run server
```

Si todo ha funcionado correctamente ya deberíamos crear correctamente películas con imágenes y acceder a ellas en el servidor de pruebas.

ViewSets y Serializers

En esta lección vamos a programar el endpoint para consultar las películas de nuestra API.

En Django generalmente necesitamos dos vistas para gestionar un modelo: una **ListView** para listar múltiples instancias y una **DetailView** para gestionar una única instancia del modelo.

Pues DRF permite combinar la lógica de ambas vistas en lo que ellos denominan un **ViewSet** o literalmente conjunto de vistas, os dejaré documentación.

Un **ViewSet** gira entorno a un modelo y su serializador, así que vamos a empezar creando unos serializadorres básicos para las película y los géneros:

films/serializers.py
```
from rest_framework import serializers
from .models import Film, FilmGenre


class FilmSerializer(serializers.ModelSerializer):
    class Meta:
        model = Film
        fields = '__all__'


class FilmGenreSerializer(serializers.ModelSerializer):
    class Meta:
        model = FilmGenre
        fields = '__all__'
```
Como véis son muy simples, les pasamos el modelo y les indicamos devuelvan todos los campos con `__all__` en los fields.

Con esto podemos crear unos **viewsets** básicos:

films/views.py
```
from rest_framework import viewsets
from .models import Film, FilmGenre
from .serializers import FilmSerializer, FilmGenreSerializer


class FilmViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Film.objects.all()
```

```
    serializer_class = FilmSerializer


class GenreViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = FilmGenre.objects.all()
    serializer_class = FilmGenreSerializer
    lookup_field = 'slug'  # identificaremos los géneros usando su
slug
```

Como véis estamos utilizando unos **viewsets** de tipo `ReadOnlyModelViewset`, cuya particularidad como podéis suponer, es que ofrecen acciones de lectura. Eso es porque no necesitamos que nuestros clientes puedan crear, editar o borrar películas y géneros, ya que son acciones protegidas sólo disponibles a los administradores de la API. Os dejaré documentación sobre este tipo de **viewset**.

Sea como sea sólo falta dar de alta los viewsets en el **router**:

server/urls.py

```
from films import views as film_views

# Api router
router = routers.DefaultRouter()
router.register('films', film_views.FilmViewSet, basename='Film')
router.register('genres', film_views.GenreViewSet,
basename='FilmGenre')
```

Y ya podremos navegar por los viewsets de nuestra API:

- http://127.0.0.1:8000/api/
- http://127.0.0.1:8000/api/films/
- http://127.0.0.1:8000/api/films/:id/
- http://127.0.0.1:8000/api/genres/:slug/

En la siguiente lección puliremos los serializadores para ofrecer más información.

Serializadores anidados

Hay algunas cosas mejorables en nuestra API.

La primera es que cuando trabajamos con relaciones en los modelos, DRF automáticamente las serializa utilizando sus campos **ids**.

En este caso los géneros de una película forman parte de una relación **ManyToMany** y serializa los **ids** en una lista numérica:

```
"genres": [
    1
]
```

Lo ideal es proporcionar más de información y para conseguirlo podemos utilizar un sistema de serialización anidada:

films/serializers.py

```
from rest_framework import serializers
```

```
from .models import Film, FilmGenre


class FilmGenreSerializer(serializers.ModelSerializer):

    class Meta:
        model = FilmGenre
        fields = '__all__'


class FilmSerializer(serializers.ModelSerializer):

    class Meta:
        model = Film
        fields = '__all__'

    genres = FilmGenreSerializer(many=True)  # new
```

El parámetro `many=True` indica que se tiene que serializar una lista de instancias, algo evidente teniendo en cuenta que `genres` es una relación **ManyToMany**. Además hay que cambiar el orden de los serializadores, ya que no se puede hacer referencia a una clase antes de declararla

Si consultamos la API veremos que ahora las películas contienen la información anidada de los géneros en forma de lista de objetos:

```
"genres": [
    {
        "id": 1,
        "name": "Prueba",
        "slug": "prueba"
    }
],
```

Otra cosa que podemos hacer es mostrar una lista de las películas que tiene cada género.

Para conseguir esta funcionalidad hay que ser un poco más creativos, ya que por defecto nuestros géneros no contienen las películas, sino que son las películas las que contienen las referencias a los géneros.

Por suerte Django permite hacer consultas inversas en las relaciones, algo que podemos usar a nuestro favor para crear nuestro propios campo **films** en el serializador de géneros:

films/serializers.py

```
class FilmGenreSerializer(serializers.ModelSerializer):

    class Meta:
        model = FilmGenre
        fields = '__all__'

    films = FilmSerializer(many=True, source="film_genres") # query reversa
```

Sin embargo hay un error en esta lógica: no podemos hacer referencia a la clase `FilmSerializer` porque se encuentra debajo de `FilmGenreSerializer`.

Para solucionar este error podemos definir un serializador anidado de películas dentro de él mismo:

films/serializers.py
```
class FilmGenreSerializer(serializers.ModelSerializer):

    class Meta:
        model = FilmGenre
        fields = '__all__'

    class NestedFilmSerializer(serializers.ModelSerializer):

        class Meta:
            model = Film
            fields = '__all__'

    films = NestedFilmSerializer(
        many=True, read_only=True)
```

Esta estructura nos lleva nuevamente a hacer referencia a `FilmGenreSerializer`, pero como se encuentra abajo del todo no podemos acceder... así que vamos a crear una vez más otro serializador anidado dentro del serializador anidado, lo cuál generará una estructura de subclases muy interesante a la par que confusa:

films/serializers.py
```
from rest_framework import serializers
from .models import Film, FilmGenre


class FilmGenreSerializer(serializers.ModelSerializer):

    class Meta:
        model = FilmGenre
        fields = '__all__'

    class NestedFilmSerializer(serializers.ModelSerializer):

        class Meta:
            model = Film
            fields = '__all__'

        class NestedFilmGenreSerializer(serializers.ModelSerializer):

            class Meta:
                model = FilmGenre
                fields = '__all__'

        genres = NestedFilmGenreSerializer(many=True)

    films = NestedFilmSerializer(
        many=True, source="film_genres")  # query reversa
```

Es un poco complejo de entender pero os aseguro que gracias a la serialización anidada se pueden hacer maravillas con muy poco código.

En cualquier caso podemos consultar la API y notar como los géneros nos devuelven un campo **films** con una lista de películas y toda su información:

```
[
```

```
  {
    "id": 1,
    "name": "Prueba",
    "slug": "prueba",
    "films": [
      {
        "id": "xxx",
        "genres": [
          {
            "id": 1,
            "name": "Prueba",
            "slug": "prueba"
          }
        ],
        "title": "Prueba de pelicula",
        "year": 2000,
        "review_short": "",
        "review_large": "",
        "trailer_url": null,
        "image_thumbnail":
"http://localhost:8000/media/films/xxx/yyy.png",
        "image_wallpaper":
"http://localhost:8000/media/films/xxx/zzz.jpg"
      }
    ]
  }
]
```


La gracia ahora es adaptar los serializadores para devolver la información que consideremos necesaria.

Por ejemplo no hace falta devolver todos los campos de la película en los géneros, por ahora será suficiente con el **id, el título, la miniatura y los géneros**:

films/serializers.py
```
class NestedFilmSerializer(serializers.ModelSerializer):

    class Meta:
        model = Film
        fields = ['id', 'title', 'image_thumbnail', 'genres']

        class NestedFilmGenreSerializer(serializers.ModelSerializer):

            class Meta:
                model = FilmGenre
                fields = '__all__'

        genres = NestedFilmGenreSerializer(many=True)
```

De hecho vamos a quitar los géneros de las películas en los géneros, es demasiado redundante y mi única intención era que vieses cómo anidar múltiples serializadores:

films/serializers.py
```
class NestedFilmSerializer(serializers.ModelSerializer):

    class Meta:
        model = Film
        fields = ['id', 'title', 'image_thumbnail']  # edited
```

```
        # BORRAMOS LO SIGUIENTE ========>

        # class
NestedFilmGenreSerializer(serializers.ModelSerializer):

        #     class Meta:
        #         model = FilmGenre
        #         fields = '__all__'

        # genres = NestedFilmGenreSerializer(many=True)
```

Así conseguiremos una serialización más simple:

```
[
  {
    "id": 1,
    "films": [
      {
        "id": "42569494-d623-446e-8d14-3686860c5277",
        "title": "Prueba de pelicula",
        "image_thumbnail":
"http://localhost:8000/media/films/xxx/yyy.png"
      }
    ],
    "name": "Prueba",
    "slug": "prueba"
  }
]
```

Sin embargo hacer todo ha desembocado en un último problemilla...

Si consultamos una película en la API veremos que nos está mostrando los géneros con las películas dentro de las películas... todo un lío:

```
[
  {
    "id": "42569494-d623-446e-8d14-3686860c5277",
    "genres": [
      {
        "id": 1,
        "films": [
          {
            "id": "42569494-d623-446e-8d14-3686860c5277",
            "title": "Prueba de pelicula",
            "image_thumbnail":
"http://localhost:8000/media/films/xxx/yyy.png"
          }
        ],
        "name": "Prueba",
        "slug": "prueba"
      }
    ],
    "title": "Prueba de pelicula",
    "year": 2000,
    "review_short": "",
    "review_large": "",
    "trailer_url": null,
    "image_thumbnail":
"http://localhost:8000/media/films/xxx/yyy.png",
```

```
    "image_wallpaper": "http://localhost:8000/media/films/xxx/zzz.jpg"
  }
]
```

Vamos a usar la misma lógica de los serializadores anidados para simplificar el serializador de géneros de las películas y que no incluya las películas:

films/serializers.py

```
class FilmSerializer(serializers.ModelSerializer):

    class Meta:
        model = Film
        fields = '__all__'

    class NestedFilmGenreSerializer(serializers.ModelSerializer):

        class Meta:
            model = FilmGenre
            fields = '__all__'

    genres = NestedFilmGenreSerializer(many=True)
```

Si probamos ahora con la nueva lógica...

```
[
  {
    "id": "42569494-d623-446e-8d14-3686860c5277",
    "genres": [
      {
        "id": 1,
        "name": "Prueba",
        "slug": "prueba"
      }
    ],
    "title": "Prueba de pelicula",
    "year": 2000,
    "review_short": "",
    "review_large": "",
    "trailer_url": null,
    "image_thumbnail":
"http://localhost:8000/media/films/xxx/yyy.png",
    "image_wallpaper": "http://localhost:8000/media/films/xxx/zzz.jpg"
  }
]
```

¡Solucionado! ¿Qué bien quedan nuestras clases anidadas verdad?

Por ahora os dejo documentación sobre los serializadores en los recursos, sólo por si queréis aprender un poco más por vuestra cuenta...

Base de datos preparada

Para acabar la unidad vamos a "instalar" la base de datos que he preparado para vosotros con mucho cariño. Un montón de buenas películas con la información e imágenes para hacer nuestros experimentos de la mejor forma posible.

Simplemente tenéis que descargar el fichero db_preparada.zip y hacer esto:

1. Parar el servidor si lo tenéis en marcha.
2. Borrar el fichero **db.sqlite3** de la raiz.
3. Borrar los directorios **migrations** de las apps **authentication** y **films**.
4. Extraer el contenido del fichero en la raíz del proyecto `server` y substutir si os lo pide.

Al final deberéis tener de nuevo la base de datos **db.sqlite3** en la raíz, así como los nuevos directorios **migrations** en las apps y un montón de carpetas en el directorio **media/films**.

Tened en cuenta que vuestros usuarios se habrán borrado, pero tendréis a vuestra disposición:

- admin@admin.com : 1234
- test@test.com : 12345678

Siempre podéis crear nuevos administradores o editar estos usuarios desde el panel de administrador.

Buen provecho.

Sistema de búsquedas y filtros en DRF

A lo largo de esta sección extenderemos nuestro `FilmViewSet` para ofrecer cuatro nuevas funcionalidades a las películas:

- **Búsqueda**: Usando un texto y buscando coincidencias.
- **Ordenación**: Ascendente o descendente a partir de varios campos.
- **Filtrado**: En base a a partir de varios campos.
- **Paginación**: Para limitar los registros por página.

Sistema de búsqueda

En esta lección añadiremos de una forma muy sencilla la opción de realizar búsquedas en varios campos del `ViewSet`.

El objetivo es dar cobertura a la típica funcionalidad de un buscador en tiempo real, dónde a partir de un texto se buscan coincidencias en la API y se devuelve la lista resultante.

Implementar esta funcionalidad en un `ViewSet` es facilísimo porque ya viene implementada y sólo hay que configurarla:

films/views.py

```
from rest_framework import viewsets, filters  # edited

class FilmViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Film.objects.all()
```

```
    serializer_class = FilmSerializer

    # Sistema de filtros
    filter_backends = [filters.SearchFilter]

    search_fields = ['title', 'year']
```

Sólo con este cambio podemos probar cómo funciona el sistema de búsquedas en el `ViewSet` utilizando el cliente gráfico de pruebas enel nuevo apartado **Filtros** que nos aparecerá en el menú superior.

Fijaros que podemos enviar tanto el título como el año, que según la estructura de las peticiones debe pasarse en un parámetro **GET** llamado **search**.

Otra cosa interesante es que podemos hacer referencia a un campo de un modelo relacionado, por ejemplo si queremos filtrar por el nombre de un género podemos hacerlo así:

```
search_fields = ['title', 'year', 'genres__name']  # edited
```

Os dejo la documentación sobre los filtros de búsqueda.

Sistema de ordenación

Al igual que el filtro de búsqueda, el filtro de ordenación viene implementado y sólo hay que activarlo:

films/views.py
```
class FilmViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Film.objects.all()
    serializer_class = FilmSerializer

    # Sistema de filtros
    filter_backends = [filters.SearchFilter, filters.OrderingFilter]
# edited

    search_fields = ['title', 'year', 'genres__name']
    ordering_fields = ['title', 'year']  # new
```

En esta ocasión debemos pasar un parámetro **GET** llamado **ordering**. Lo interesante es que por defecto el orden es descendiente, eso es de la A a la Z en textos y de menos a más en números, pero se puede negar a ascendiente añadiendo un **-** delante del campo.

Además igual que antes podemos hacer referencia a un campo de un modelo relacionado, por ejemplo para ordenar a partir del nombre los géneros pondremos:

```
ordering_fields = ['title', 'year', 'genres__name']  # edited
```

Es interesante que funcione incluso con campos `ManyToMany` sin hacer ninguna configuración extra.

Os dejo documentación por si queréis aprender más sobre los filtros de ordenamiento.

Sistema de filtros de campo

En las anteriores lecciones de esta unidad hemos configurado filtros genéricos de búsqueda y ordenamiento que ya se encuentran implementados en DRF. ¿Pero cómo podemos filtrar los resultados a partir de un género o año explícito?

Para conseguir este comportamiento debemos utilizar la librería `DjangoFilterBackend` de DRF que nos permite crear nuestros propios filtros. Ésta requiere instalar y configurar una app externa llamada **django-filter** tal como explican en la documentación oficial de DRF:

```
cd server
pipenv install django-filter
```

Luego tenemos que activarla:

server/settings.py

```
INSTALLED_APPS = [
    # Django external apps
    'corsheaders',
    'rest_framework',
    'django_rest_passwordreset',
    'django_filters',  # new
]
```

Ahora importamos `DjangoFilterBackend` y lo añadimos en la lista `filter_backends` de nuestra `ListView`:

films/views.py

```
from django_filters.rest_framework import DjangoFilterBackend  # new

class FilmViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Film.objects.all()
    serializer_class = FilmSerializer

    # Sistema de filtros
    filter_backends = [DjangoFilterBackend,  # edited
                       filters.SearchFilter, filters.OrderingFilter]

    search_fields = ['title', 'year', 'genres__name']
    ordering_fields = ['title', 'year', 'genres__name']
```

Con este **backend** activo ya podemos configurar filtros de campo con operadores relacionales y otras funcionaliadades:

```
filterset_fields = {
    'year': ['lte', 'gte'],  # Año menor o igual, mayor o igual que
    'genres': ['exact']      # Género exacto
}
```

Si vamos a la interfaz de la API podremos probar nuestro sistema de filtros de campo y analizar la estructura de las peticiones, que en este caso tendríamos **year__lte**, **year__gte** y múltiples **genres**. Lo mejor de todo es que estos filtros son acumulativos, de manera que que al final podríamos realizar consultas avanzadas en una sola petición, por ejemplo:

- Año mayor que 1980: `year__gte=1980`

- Año menor que 2000: `year__lte=2000`
- Género Fantasía: `genres=3`
- Género Crimen: `genres=12`
- Ordenadas por año: `ordering=year`

Que en una sola query quedaría así:

```
?year__lte=2000&year__gte=1980&genres=3&genres=12&ordering=title
```

Como véis en conjunto hemos implementado un potente sistema de búsquedas y filtros con muy poco código... Si es que DRF es una maravilla.

Sistema de paginación

La guinda del pastel de la `FilmViewSet` la pondremos con un sistema de paginación, el cuál se utiliza para limitar los registros devueltos a las peticiones de los clientes, evitando de esa forma saturar al servidor con consultas inmensas.

Según la documentación que os dejo en los recursus, podemos activar la paginación por defecto en todas las vistas o configurarla manualmente donde queramos. Nosotros vamos a hacerlo de la segunda forma porque no me interesa paginar los géneros automáticamente.

Sólo tenemos que importar la clase del paginador genérico y asignarla al `ViewSet`:

films/views.py

```
from rest_framework.pagination import PageNumberPagination  # new

class FilmViewSet(viewsets.ReadOnlyModelViewSet):
    # ...

    # Sistema de paginación
    pagination_class = PageNumberPagination
    pagination_class.page_size = 8  # películas por página
```

Como veremos en la interfaz de la API ahora por defecto tenemos limitados los registros y podemos navegar entre las diferentes páginas que nos aparecen en la parte superior.

Fijaros que ahora las películas nos aparecen dentro del campo **results** de un objeto que contiene información del paginador, como por ejemplo **count** con el número total de registros, **next** y **previous** con los enlaces a la siguiente y anterior página. Además la nevagación como tal se gestiona con un parámetro de tipo **GET** llamado **page** que indica la página actual.

Lo genial de este paginador es que funciona automáticamente incluso con los filtros activos, por ejemplo si buscamos las películas del año 1990 en adelante y ordenadas por año:

```
?ordering=year&year__gte=1990
```

Simplemente genial.

Paginación personalizada

En la última lección de la unidad os voy a enseñar a personalizar la paginación de la API para cubrir todas las necesidades que un cliente pueda tener.

Para personalizar la paginación, la documentación nos explica que debemos crear nuestra propia clase a partir de `PageNumberPagination` y extender su método `get_paginated_response`:

films/views.py

```
from rest_framework.response import Response  # new

class ExtendedPagination(PageNumberPagination):
    page_size = 8

    def get_paginated_response(self, data):

        return Response({
            'count': self.page.paginator.count,
            'num_pages': self.page.paginator.num_pages,
            'page_number': self.page.number,
            'page_size': self.page_size,
            'next_link': self.get_next_link(),
            'previous_link': self.get_previous_link(),
            'results': data
        })


class FilmViewSet(viewsets.ReadOnlyModelViewSet):
    # ...

    # Sistema de paginación
    pagination_class = ExtendedPagination  # edited
```

Usando nuestro propio paginador estamos proveyendo de mucha más información al cliente:

- `count`: El número de registros
- `num_pages`: El número de páginas
- `page_number`: El número de página actual
- `page_size`: El número de página actual
- `next_link`: El enlace a la siguiente página
- `previous_link`: El enlace a la anterior página

Otra cosa que podemos hacer, y de hecho vamos a hacer por petición de mi compañero, es modificar estos campos a voluntad, concretamente los enlaces para navegar en la paginación.

Me han pedido si era posible dejar sólo los parámetros de la consulta, ya que el plugin de React que va a utilizar los requiere de esta forma, así que vamos a hacerlo:

films/views.py

```
class ExtendedPagination(PageNumberPagination):
    page_size = 8

    def get_paginated_response(self, data):
```

```
        # Recuperamos los valores por defecto
        next_link = self.get_next_link()
        previous_link = self.get_previous_link()

        # Hacemos un split en la primera '/' dejando sólo los
parámetros
        if next_link:
            next_link = next_link.split('/')[-1]

        if previous_link:
            previous_link = previous_link.split('/')[-1]

        # Modificamos los valores devueltos
        return Response({
            'count': self.page.paginator.count,
            'num_pages': self.page.paginator.num_pages,
            'page_number': self.page.number,
            'page_size': self.page_size,
            'next_link': next_link,            # edited
            'previous_link': previous_link,    # edited
            'results': data
        })
```

Y con este cambio acabamos la unidad.

Sistema de perfiles de usuario en DRF

En esta unidad vamos a extender el modelo de usuario agregándole un campo para guardar la imagen de su **avatar**. También implementaremos una nueva `APIView` para gestionar el perfil y permitir al usuario modificar el nick y el avatar.

Campo avatar

Añadimos el avatar al modelo:

authentication/models.py
```
def path_to_avatar(instance, filename):                    # new
    return f'avatars/{instance.id}/{filename}'             # new


class CustomUser(AbstractUser):
    # ...
    avatar = models.ImageField(                            # new
        upload_to=path_to_avatar, null=True, blank=True)   # new
```

Migramos la app:

```
cd server
pipenv run makemigrations authentication
pipenv run migrate authentication
```

Añadimos el campo al serializador:

authentication/serializers.py
```
class UserSerializer(serializers.ModelSerializer):
```

```
    email = serializers.EmailField(
        required=True)
    username = serializers.CharField(
        required=True)
    password = serializers.CharField(
        min_length=8, write_only=True)
    avatar = serializers.ImageField(    # new
        required=False)                 # new

    class Meta:
        model = get_user_model()
        fields = ('email', 'username', 'password', 'avatar')  # edited
```

Con esto ya tenemos el campo **avatar**. Si vamos a la interfaz web de la API y registramos un nuevo usuario veréis que podemos adjuntar una imagen. Probad a registrar un usuario con un avatar:

```
Email: avatar@avatar.com
Username: avatar
Password: 12345678
Avatar: Imagen que queramos
```

Veréis que funciona bien, el usuario se crea y la imagen se guarda. Pero hay algo interesante, y es que si comprobamos la ruta donde se almacena la imagen, en lugar del **id** tenemos **None**. Esto sucede porque el **id** del usuario no existe en el momento que se registra el usuario. Este escenario se menciona explícitamente en la documentación del parámetro **upload_to**:

```
In most cases, this object will not have been saved to the database
yet, so if it uses the default AutoField, it might not yet have a
value for its primary key field.
```

Como la clase **User** utiliza efectivamente el **AutoField** incremental para almacenar el **id**, éste no existe. Esto explicaría también porque en el modelo película este error no sucede, pues allí utilizamos un **UUIDField** en lugar del **AutoField** por defecto.

En cualquier caso en la práctica no es algo que deba preocuparnos, porque cuando el usuario quiera modificar su avatar deberá haberse registrado antes.

Vista de perfil

Para gestionar nuestro perfil debemos proporcionar dos funcionalidades nueva, la de consulta de la información y la de actualización.

DRF cuenta con una vista llamada `RetrieveUpdateAPIView` pensada para tareas de lectura y actualización, os dejo el enlace a la documentación. Ahí veremos que la vista implementa tres métodos: **GET, POST y PATCH**. Según la documentación de Mozilla sobre los métodos de las peticiones HTTP:

- El método `GET` solicita una representación de un recurso específico. Las peticiones que usan el método GET sólo deben recuperar datos.

- El método `POST` se utiliza para enviar una entidad a un recurso en específico, causando a menudo un cambio en el estado o efectos secundarios en el servidor.
- El método `PATCH` es utilizado para aplicar modificaciones parciales a un recurso.

Así que vamos a utilizar **GET** para devolver la información de lectura y **PATCH** para manejar la actualización parcial del usuario.

Empezamos creando la vista, asignando el serializador y los métodos que queremos permitir, en nuestro caso manejaremos el usuario así que pasaremos `UserSerializer` y los métodos `GET` y `PATCH`:

authentication/views.py
```
class ProfileView(generics.RetrieveUpdateAPIView):
    serializer_class = UserSerializer
    http_method_names = ['get', 'patch']
```

Añadimos la vista a las URL:

authentication/urls.py
```
from .views import LoginView, LogoutView, SignupView, ProfileView  # edited

urlpatterns = [

    # ...

    # Profile views
    path('user/profile/',
         ProfileView.as_view(), name='user_profile'),
]
```

Si nos identificamos `http://localhost:8000/api/auth/login/` y accdemos acceder al perfil `http://localhost:8000/api/user/profile/` obtendremos un error:

```
'ProfileView' should either include a `queryset` attribute, or override the `get_queryset()` method.
```

Esto sucede porque la vista no sabe con qué información rellenar el serializador.

Nosotros queremos rellenarlo con la información del usuario, así que podemos hacer uso del método `get_object`, la versión para un único objeto del `get_queryset` y que tome el usuario de la propia petición:

authentication/views.py
```
class ProfileView(generics.RetrieveUpdateAPIView):
    serializer_class = UserSerializer
    http_method_names = ['get', 'patch']

    def get_object(self):
        if self.request.user.is_authenticated:
            return self.request.user
```

Si intentamos ahora acceder al perfil ya veremos la información del usuario, sin embargo al modificar algún dato y presionar **PATCH** nos devolverá un error:

```
"avatar": [ "Este campo no puede ser nulo."]
```

Al parecer que `required=False` no es suficiente para quee el avatar puede ser nulo y tenemos que especificarlo manualmente así:

authentication/serializers.py

```
avatar = serializers.ImageField(
    required=False, allow_null=True)  # edited
```

Ahora sí deberíamos ser capaces de editar los campos del perfil `http://localhost:8000/api/user/profile/`.

```
{
  "email": "test@test.com",
  "username": "test",
  "avatar": null
}
```

Pero no vamos a dejarlo así, hay varias cosas que debemos arreglar.

Puliendo el serializador

Como os decía debemos arreglar algunas cosas, empezando por el hecho de que podemos editar el email y eso no entra en mis planes. En esta API por ahora sólo quiero dar la opción de editar el nombre de usuario, que debe ser único, y el avatar.

Una forma simple de bloquear la edición de un campo es sobreescribir el método **update** del serializador y borrarlo del diccionario de campos validados:

```
def update(self, instance, validated_data):
    validated_data.pop('email', None)                # prevenimos el
borrado
    return super().update(instance, validated_data)  # seguimos la
ejecución
```

Como dato el método **create** de un serializador se ejecuta una única vez durante la creación de la instancia, mientras que el **update** se ejecuta siempre que actualizamos algo.

En cualquier caso siguamos mejorando nuestra API, por ejemplo editando el nombre de usuario y poniendo uno de un usuario que ya exista:

```
UNIQUE constraint failed: authentication_customuser.username
```

Como vemos se devuelve un error muy feo. Si el **DEBUG** estubiera desactivado se devolvería un código 500 indicando un error interno del servidor.

Podemos añadir nuestro propio validador para el campo **username** y devolver un error en condiciones sin que explote el servidor por el camino:

authentication/serializers.py

```
# ...

def validate_password(self, value):
    return make_password(value)

def validate_username(self, value):
    value = value.replace(" ", "")  # Ya que estamos borramos los
espacios
```

```
        try:
            user = get_user_model().objects.get(username=value)
            # Si es el mismo usuario mandando su mismo username le dejamos
            if user == self.instance:
                return value
        except get_user_model().DoesNotExist:
            return value
        raise serializers.ValidationError("Nombre de usuario en uso")
```

Con esta validación mucho más elegante el servidor devolverá un código **400 Bad Request** con información detallada del error:

```
"username": ["Nombre de usuario en uso"]
```

En este punto no perdemos nada por agregar otra validación al email, ya que durante el registro si se utiliza un email en uso sucede lo mismo que antes y explota devolviendo un código 500 sin nada de información:

authentication/serializers.py

```
def validate_email(self, value):
    # Hay un usuario con este email ya registrado?
    try:
        user = get_user_model().objects.get(email=value)
    except get_user_model().DoesNotExist:
        return value
    # En cualquier otro caso la validación fallará
    raise serializers.ValidationError("Email en uso")
```

Con esto ya tenemos nuestro serializador está perfectamente validado y acabamos la unidad.

istema de películas de usuario en DRF

En esta última unidad vamos a programar la interacción entre los usuarios y las películas. Esto nos permitirá que puedan gestionar diferentes opciones en las películas:

- Marcarlas como favoritas
- Configurar estados (vista, por ver...)
- Añadir una puntuación
- Escribir reseñas

Además esto implicará que las películas deberán mostrar el número de favoritos y la nota media en su endpoint, por lo que deberemos configurar estos campos.

Modelo de películas de usuario

Para poder gestionar todo el tinglado necesitamos dar de alta un nuevo modelo para manejar la relación entre usuario y película, lo llamaremos FilmUser:

server/films/models.py

```
from django.conf import settings  # new
from django.core.validators import MaxValueValidator  # new

class FilmUser(models.Model):

    STATUS_CHOICES = (
        (0, "Sin estado"),
        (1, "Vista"),
        (2, "Quiero verla"))

    user = models.ForeignKey(settings.AUTH_USER_MODEL,
                             on_delete=models.CASCADE)
    film = models.ForeignKey(Film, on_delete=models.CASCADE)

    # Se podría hacer en tres modelos separados para que sea más
eficiente
    # pero a nivel de desarrollo habría que hacer lo mismo tres veces

    state = models.PositiveSmallIntegerField(
        choices=STATUS_CHOICES, default=0)  # Al crearse sin estado se
borra
    favorite = models.BooleanField(
        default=False)
    note = models.PositiveSmallIntegerField(
        null=True, validators=[MaxValueValidator(10)])
    review = models.TextField(null=True)

    class Meta:
        unique_together = ['film', 'user']
        ordering = ['film__title']
```

Migramos:

```
pipenv run makemigrations
pipenv run migrate
```

Ahora vamos a crear una vista de tipo `APIView` para manejar estas interacciones, pero antes necesitamos crear un serializador para las películas de usuario:

server/films/serializers.py

```
from .models import Film, FilmGenre, FilmUser  #updated

class FilmUserSerializer(serializers.ModelSerializer):

    film = FilmSerializer(read_only=True)

    class Meta:
        model = FilmUser
        fields = ['film', 'favorite', 'note', 'state', 'review']
```

Y ahora la viewset para nuestro modelo `FilmUser`:

server/films/views.py

```
from rest_framework import viewsets, filters, status, views  # updated
from .models import Film, FilmGenre, FilmUser  # updated
from .serializers import (FilmSerializer, FilmGenreSerializer,
                          FilmUserSerializer)  # updated

class FilmUserViewSet(views.APIView):

    # El método GET devolverá las peliculas del usuario
    def get(self, request, *args, **kwargs):
        queryset = FilmUser.objects.filter(user=self.request.user)
```

```
        serializer = FilmUserSerializer(queryset, many=True)
        return Response(serializer.data, status=status.HTTP_200_OK)

    # El método POST permitirá gestionar su información de la película
    def post(self, request, *args, **kwargs):
        try:
            film = Film.objects.get(id=request.data['uuid'])
        except Film.DoesNotExist:
            return Response(
                {'status': 'Film not found'},
                status=status.HTTP_404_NOT_FOUND)

        # Una vez recuperada la película creamos o recuperamos su
FilmUser
        film_user, created = FilmUser.objects.get_or_create(
            user=request.user, film=film)

        # Configuramos cada campo
        film_user.state = request.data.get('state', 0)
        film_user.favorite = request.data.get('favorite', False)
        film_user.note = request.data.get('note', -1)
        film_user.review = request.data.get('review', None)

        # Si se marca la pelicula como NO VISTA la borramos
automáticamente
        if int(film_user.state) == 0:
            film_user.delete()
            return Response(
                {'status': 'Deleted'}, status=status.HTTP_200_OK)

        # En otro caso guardamos los campos de la película de usuario
        else:
            film_user.save()

        return Response(
            {'status': 'Saved'}, status=status.HTTP_200_OK)
```

Damos de alta la URL en un nuevo endpoint:

server/server/urls.py

```
urlpatterns = [
    # ...
    path('api/userfilms/', film_views.FilmUserViewSet.as_view())
]
```

En este punto deberíamos ser capaces de acceder a `http://127.0.0.1:8000/api/userfilms/` y ver las películas del usuario identificado en el administrador.
Como no hemos creado ninguna esa respuesta estará vacía, pero podemos crear una película de usuario de prueba haciendo una petición `POST`.

Por ejemplo para añadir la información de la película de **Apocalypse Now**, cuyo **uuid** es **d1d5acd7-5f76-4faa-aca7-fdb9cc88eb02** escribiremos lo siguiente:

```
{    "uuid": "d1d5acd7-5f76-4faa-aca7-fdb9cc88eb02",
    "favorite": 1,
    "note": 10,
    "state": 1,
    "review": "Me encanta esta película..."
```

```
}
```


Esto nos devolverá:


```
{
    "status": "Saved"
}
```


Indicando que el registro de la película de usuario se ha guardado correctamente.

Podemos consultar de nuevo la lista recargando el endpoint y ahora sí veremos la lista con las películas favoritas del usuario:


```
[
    {
        "film": {
            "id": "d1d5acd7-5f76-4faa-aca7-fdb9cc88eb02",
            "genres": [
                {
                    "id": 15,
                    "name": "Bélica",
                    "slug": "belica"
                },
                {
                    "id": 3,
                    "name": "Drama",
                    "slug": "drama"
                },
                {
                    "id": 10,
                    "name": "Misterio",
                    "slug": "misterio"
                }
            ],
            "title": "Apocalypse Now",
            "year": 1979,
            "review_short": "En 1969, durante la guerra de Vietnam, el
coronel Kurtz (Marlon Brando), de las Fuerzas Especiales del Ejército
de Estados Unidos, se ha vuelto loco y ahora manda a sus propias
tropas de montañeses, dentro de la neutral Camboya, como un
semidiós.",
            "review_large": "En 1969, durante la guerra de Vietnam, el
coronel Kurtz (Marlon Brando), de las Fuerzas Especiales del Ejército
de Estados Unidos, se ha vuelto loco y ahora manda a sus propias
tropas de montañeses, dentro de la neutral Camboya, como un semidiós.
\r\n\r\nEl coronel Lucas (Harrison Ford) y el general Corman (G. D.
Spradlin), cada vez más preocupados por las operaciones de vigilancia
de Kurtz, asignan al capitán de MACV-SOG Benjamin L. Willard (Martin
Sheen) para que «ponga fin» a Kurtz «con extremo perjuicio»
(asesinarlo).",
            "trailer_url": "https://www.youtube.com/watch?v=9l-ViOOFH-
s",
            "image_thumbnail":
"http://localhost:8000/media/films/d1d5acd7-5f76-4faa-aca7-
fdb9cc88eb02/17.jpg",
            "image_wallpaper":
"http://localhost:8000/media/films/d1d5acd7-5f76-4faa-aca7-
fdb9cc88eb02/17-1.jpg"
        },
```

```
        "favorite": true,
        "note": 10,
        "state": 1,
        "review": "Me encanta esta película..."
    }
]
```

Si queremos modificar la película de usuario basta con enviar de nuevo los campos deseados a la petición:

```
{   "uuid": "d1d5acd7-5f76-4faa-aca7-fdb9cc88eb02",
    "favorite": 0,
    "note": 0,
    "state": 1,
    "review": "Ya no me gusta esta película"
}
```

Pero si enviamos una petición sin estado, o con el estado en cero (que es de película no vista), la película de usuario se borrará de la base de datos:

```
{
    "uuid": "d1d5acd7-5f76-4faa-aca7-fdb9cc88eb02"
}
```

Devolviendo:

```
{
    "status": "Deleted"
}
```

Evidentemente esto es solo mi forma de manejar toda la lógica, cada uno puede decidir como y cuando borrar los campos.

Reseñas y películas favoritas

Ahora que manejamos favoritos y notas de película, podemos almacenarlas dinámicamente en las películas. Para ello vamos a crear dos nuevos campos en el modelo de película:

server/films/models.py
```
from django.db.models import Sum  # new
from django.db.models.signals import post_save  # new

class Film(models.Model):

    # ...

    # Estadisticas actualizadas con señales
    favorites = models.IntegerField(
        default=0, verbose_name="favoritos")
    average_note = models.FloatField(
        default=0.0, verbose_name="nota media",
        validators=[MaxValueValidator(10.0)])
```

Para actualizar estos campos lo haremos usando una señal `post_save` a nivel de `FilmUser`:

server/films/models.py
```
def update_film_stats(sender, instance, **kwargs):
```

```
    # Actualizamos los favoritos contando los favoritos de esa
película
    count_favorites = FilmUser.objects.filter(
        film=instance.film, favorite=True).count()
    instance.film.favorites = count_favorites
    # Actualizamos la nota recuperando el número de notas y haciendo
la media
    notes = FilmUser.objects.filter(
        film=instance.film).exclude(note__isnull=True)
    count_notes = notes.count()
    sum_notes = notes.aggregate(Sum('note')).get('note__sum')
    # Intentamos hacer la media con dos decimales usando un try
    # Fallará si sum_notes es None como count_notes es 0
    # Esto sucede las primeras veces porque aún no hay notas
establecidas
    try:
        instance.film.average_note = round(sum_notes/count_notes, 2)
    except:
        pass
    # Guardamos la película
    instance.film.save()

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

Migramos:

```
pipenv run makemigrations
pipenv run migrate
```

Ahora podemos añadir los campos favorites y average_note como campos de ordenación en el viewset:

server/films/views.py

```
class FilmViewSet(viewsets.ReadOnlyModelViewSet):
    # ...
    ordering_fields = ['title', 'year',
                       'genres__name', 'favorites', 'average_note']
```

Y si creamos una petición de película de usuario en `http://127.0.0.1:8000/api/userfilms/`, por ejemplo la de antes:

```
{    "uuid": "d1d5acd7-5f76-4faa-aca7-fdb9cc88eb02",
    "favorite": 1,
    "note": 10,
    "state": 1,
    "review": "Me encanta esta película..."
}
```

Deberíamos ser capaces de ver la información de esta película en su viewset con la nota media y el número de favoritos `http://127.0.0.1:8000/api/films/d1d5acd7-5f76-4faa-aca7-fdb9cc88eb02/`:

```
{
    "id": "d1d5acd7-5f76-4faa-aca7-fdb9cc88eb02",
    // ...
    "favorites": 1,
    "average_note": 10.0
}
```

¡Perfecto! Ya tenemos el viewset de `UserFilm` casi listo y el proyecto acabado, solo falta un pequeñísimo detalle.

Proteger las urls con interacción de usuario

Si intentamos acceder al endpoint de `UserFilm` sin estar autenticados `http://127.0.0.1:8000/api/userfilms/` aparece un error `TypeError at /api/userfilms/`.

Podemos mejorar la seguridad estableciendo un requisito en la `APIView` para requerir autenticación en esta vista, es muy sencillo:

server/films/views.py

```
from rest_framework import (viewsets, filters, status, views,
                            authentication, permissions) # updated

class FilmUserViewSet(views.APIView):
    authentication_classes = [authentication.SessionAuthentication]  # new
    permission_classes = [permissions.IsAuthenticated] # new
```

Ahora al acceder sin estar autenticado a `http://127.0.0.1:8000/api/userfilms/` nos devolverá el siguiente error, pero en esta ocasión a nivel de API, evitando el fallo de código:

```
{
    "detail": "Las credenciales de autenticación no se proveyeron."
}
```

Y con esto acabamos este curso fallido de Django Rest Framework.

Si os ha gustado este curso y aún no sois alumnos de mi curso de Django en Udemy considerad adquirirlo en este enlace, en él explico todas las claves de este framework web.