Introducción a Django y Django Rest Framework (DRF)

Parte 1: Introducción a Django y Django Rest Framework (DRF)

¿Qué es Django?

Django es un framework de desarrollo web de alto nivel escrito en Python que permite construir aplicaciones robustas y escalables rápidamente. Django sigue el principio **"Don't Repeat Yourself" (DRY)**, lo cual facilita el mantenimiento del código. Está diseñado para hacer más fácil el desarrollo de aplicaciones con una estructura limpia y organizada.

Características clave de Django:

- **Estructura modular**: Organiza el proyecto en aplicaciones reutilizables, lo que permite segmentar y gestionar diferentes funcionalidades de la aplicación.
- **ORM (Object-Relational Mapping)**: Incluye un sistema ORM que simplifica la interacción con la base de datos usando Python en lugar de SQL.
- **Seguridad**: Ofrece protección contra ataques comunes, como inyección de SQL, CSRF y XSS.
- **Escalabilidad**: Es ideal para aplicaciones que necesitan manejar una gran cantidad de tráfico y datos.

Estructura Básica de Django

Cuando se crea un proyecto Django, la estructura típica incluye:

1. **Carpeta del proyecto**: Contiene la configuración general de Django, como settings.py, urls.py, y wsgi.py.
2. **Aplicaciones (apps)**: Componentes individuales del proyecto que pueden manejar diferentes funcionalidades. Cada app incluye sus propios archivos models.py, views.py, y admin.py.
3. **Carpeta de templates**: Ubicada dentro de las apps o en una carpeta global de templates, contiene los archivos HTML de la aplicación.
4. **Archivos de migración**: Permiten actualizar la estructura de la base de datos de acuerdo a los cambios realizados en los modelos.

Comando para crear un proyecto: Para crear un proyecto nuevo en Django, usa el siguiente comando:

```
django-admin startproject nombre_del_proyecto
```

Comando para crear una app: Dentro del proyecto, puedes crear una app con:

```
python manage.py startapp nombre_de_la_app
```

¿Qué es Django Rest Framework (DRF)?

Django Rest Framework es una biblioteca potente y flexible que se integra con Django para crear APIs RESTful. DRF facilita el desarrollo de APIs al proporcionar herramientas para:

- **Serializar datos**: Transformar datos complejos de Django (como objetos de modelos) en JSON u otros formatos que se pueden transmitir por HTTP.
- **Manejo de vistas y rutas**: Crear vistas para operaciones CRUD y definir endpoints de API.
- **Autenticación y permisos**: Controlar el acceso a la API mediante autenticación y permisos configurables.

Beneficios de usar DRF:

- **Eficiencia**: DRF permite desarrollar APIs de manera eficiente con un enfoque modular y escalable.
- **Compatibilidad**: Se integra fácilmente con aplicaciones Django existentes.
- **Soporte para API RESTful completo**: Incluye manejo de serialización, autenticación y permisos.

Instalación de Django y DRF

Para instalar Django y DRF, sigue estos pasos:

1. **Crear un entorno virtual**: Es recomendable usar un entorno virtual para aislar las dependencias del proyecto.

```
python -m venv env
source env/bin/activate   # Para MacOS/Linux
```

```
env\Scripts\activate      # Para Windows
```

2. **Instalar Django**: Instala Django mediante pip.

```
pip install django
```

3. **Instalar Django Rest Framework**: Una vez instalado Django, instala DRF:

```
pip install djangorestframework
```

4. **Iniciar un proyecto Django**: Después de instalar Django, crea el proyecto:

```
django-admin startproject mi_proyecto
```

5. **Crear una app para la API**: Dentro del proyecto, crea una app donde construiremos los endpoints de la API:

```
cd mi_proyecto
python manage.py startapp mi_api
```

Configuración de Django Rest Framework en el Proyecto

Para activar DRF en el proyecto, debemos agregarlo a la configuración del proyecto.

1. **Modificar settings.py**: Abre el archivo settings.py y añade 'rest_framework' en la lista de INSTALLED_APPS:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
```

```
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',  # Activar Django Rest Framework
    'mi_api',          # Nuestra app de API
]
```

2. **Configuración adicional (opcional)**: Puedes configurar algunos ajustes de DRF en settings.py para manejar aspectos como permisos globales y paginación:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.IsAuthenticated',
    ],
    'DEFAULT_PAGINATION_CLASS':
'rest_framework.pagination.PageNumberPagination',
    'PAGE_SIZE': 10
}
```

- **DEFAULT_PERMISSION_CLASSES** define la clase de permiso predeterminada para todas las vistas de la API.
- **DEFAULT_PAGINATION_CLASS** y **PAGE_SIZE** configuran la paginación para los resultados de la API.

Comprobar la configuración inicial

1. **Migraciones iniciales**: Antes de ejecutar el proyecto, aplica las migraciones para crear las tablas iniciales en la base de datos:

```
python manage.py migrate
```

2. **Ejecutar el servidor**: Ahora, ejecuta el servidor de desarrollo para comprobar que la configuración funciona correctamente:

```
python manage.py runserver
```

3. **Verificación**: Abre el navegador y accede a http://127.0.0.1:8000/. Deberías ver la página inicial de Django, lo que indica que el servidor está funcionando.

Resumen de la Configuración Inicial

En esta parte hemos cubierto:

- **Estructura básica de Django**: Comprensión de la arquitectura del proyecto y cómo crear aplicaciones.
- **Introducción a DRF**: Cómo facilita la creación de APIs RESTful en Django.
- **Instalación y configuración de DRF**: Pasos para instalar DRF y configurarlo en un proyecto Django.
- **Verificación de la configuración**: Ejecutar el servidor y comprobar que todo está listo para empezar a crear endpoints de API.

Construcción de una API de Gestión de Tareas con Django y DRF

---

1. Configuración Inicial del Proyecto y la App

Paso 1: Crear un Proyecto Django y una App para la API

1. **Crear el entorno virtual e instalar Django y DRF**:

```
python -m venv env
source env/bin/activate  # MacOS/Linux
env\Scripts\activate     # Windows
pip install django djangorestframework
```

2. **Crear el proyecto Django**:

```
django-admin startproject gestion_tareas
cd gestion_tareas
```

3. **Crear la app tareas para nuestra API**:

```
python manage.py startapp tareas
```

Paso 2: Configurar Django Rest Framework en settings.py

1. Abre el archivo settings.py en gestion_tareas/gestion_tareas.
2. Agrega 'rest_framework' y 'tareas' a INSTALLED_APPS:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
```

```
    'tareas',
]
```

3. **Configura DRF en settings.py (opcional)**:

```
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.AllowAny',
    ],
}
```

4. **Migraciones iniciales**:

```
python manage.py migrate
```

2. Crear el Modelo de Tareas

1. Abre el archivo models.py dentro de la carpeta tareas.
2. Define un modelo para la entidad Tarea:

```
from django.db import models

class Tarea(models.Model):
    titulo = models.CharField(max_length=100)
    descripcion = models.TextField(blank=True)
    completada = models.BooleanField(default=False)
    fecha_creacion = models.DateTimeField(auto_now_add=True)

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

3. **Migrar el modelo a la base de datos**:

```
python manage.py makemigrations
python manage.py migrate
```

Este modelo Tarea tiene:

- titulo: Nombre de la tarea.
- descripcion: Descripción opcional de la tarea.
- completada: Estado de la tarea (completada o no).
- fecha_creacion: Fecha y hora en que se creó la tarea.

---

3. Crear un Serializador para el Modelo de Tareas

1. Crea un archivo llamado serializers.py dentro de la carpeta tareas.
2. Define el serializador para el modelo Tarea:

```
from rest_framework import serializers
from .models import Tarea

class TareaSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tarea
        fields = ['id', 'titulo', 'descripcion', 'completada', 'fecha_creacion']
```

El **serializador** TareaSerializer convierte los objetos Tarea a formato JSON y viceversa, facilitando la transferencia de datos a través de la API.

4. Crear las Vistas de la API

1. Abre el archivo views.py dentro de la carpeta tareas.
2. Crea las vistas para manejar las operaciones CRUD de las tareas:

```
from rest_framework import viewsets
from .models import Tarea
from .serializers import TareaSerializer

class TareaViewSet(viewsets.ModelViewSet):
    queryset = Tarea.objects.all().order_by('-fecha_creacion')
    serializer_class = TareaSerializer
```

Aquí, TareaViewSet es un conjunto de vistas (ViewSet) que proporciona automáticamente todas las operaciones CRUD (listar, crear, actualizar, eliminar) para el modelo Tarea.

5. Configurar las Rutas de la API

1. Crea un archivo llamado urls.py dentro de la carpeta tareas.
2. Define las rutas para el ViewSet TareaViewSet usando un enrutador:

```
from django.urls import path, include
from rest_framework.routers import DefaultRouter
from .views import TareaViewSet

router = DefaultRouter()
router.register(r'tareas', TareaViewSet)

urlpatterns = [
```

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

3. **Incluir las rutas de la app tareas en el archivo principal de rutas urls.py del proyecto**:

- Abre gestion_tareas/urls.py y añade la ruta de tareas:

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

urlpatterns = [
  path('admin/', admin.site.urls),
  path('api/', include('tareas.urls')),  # Ruta de la API de tareas
]
```

6. Probar la API

Ejecuta el servidor de desarrollo para probar la API:

```
python manage.py runserver
```

Endpoints Disponibles

- GET /api/tareas/: Listar todas las tareas.
- POST /api/tareas/: Crear una nueva tarea.
- GET /api/tareas/<id>/: Obtener una tarea específica.
- PUT /api/tareas/<id>/: Actualizar una tarea específica.
- DELETE /api/tareas/<id>/: Eliminar una tarea específica.

Abre el navegador y accede a http://127.0.0.1:8000/api/tareas/. Deberías ver una interfaz de DRF que te permite probar los endpoints.

Resumen

Hemos creado una **API de Gestión de Tareas** en Django y DRF que permite realizar todas las operaciones CRUD. Este ejemplo cubre:

- **Configuración de un proyecto Django con DRF**.
- **Definición de un modelo Tarea**.
- **Creación de un serializador TareaSerializer** para transformar los datos.
- **Configuración de las vistas usando TareaViewSet**.
- **Definición de rutas** para exponer los endpoints de la API.

Esta API puede conectarse fácilmente con un frontend como React

Parte 2: Creación de APIs con Serializers y Views

Ya que tenemos una estructura de API básica con un modelo Tarea, en esta sección vamos a detallar cómo funcionan los **Serializers** y las **Views**, y cómo configurar manualmente los endpoints CRUD en caso de que no se usen ViewSets.

---

1. Profundizando en los Serializers

En Django Rest Framework, un **serializer** convierte datos complejos (como objetos de modelos de Django) en datos nativos de Python que pueden ser fácilmente convertidos a JSON para su transmisión a través de la API. Los serializers también validan y procesan datos recibidos de peticiones HTTP antes de guardarlos en la base de datos.

Código del Serializador TareaSerializer

Nuestro TareaSerializer actualmente se ve así:

```
from rest_framework import serializers

from .models import Tarea
```

```
class TareaSerializer(serializers.ModelSerializer):
    class Meta:
        model = Tarea
        fields = ['id', 'titulo', 'descripcion', 'completada', 'fecha_creacion']
```

Explicación Detallada

- **serializers.ModelSerializer**: Es un tipo de serializer que simplifica el proceso de serialización, ya que genera automáticamente los campos en base al modelo.
- **Meta**: La clase interna Meta define la configuración del serializer, incluyendo el modelo y los campos que queremos exponer en la API.
    - **model**: Especifica el modelo de Django (Tarea) que estamos serializando.
    - **fields**: Lista los campos que queremos incluir en la representación JSON.

Con el TareaSerializer, cada instancia de Tarea se convierte automáticamente en un objeto JSON que incluye id, titulo, descripcion, completada y fecha_creacion.

2. Creación Manual de Views para Operaciones CRUD

Hasta ahora, hemos utilizado ModelViewSet en views.py para proporcionar las operaciones CRUD automáticamente. Ahora vamos a configurar manualmente las **vistas** para cada operación CRUD usando **API views**.

Archivo: views.py (Manual CRUD con API views)

Modifica views.py para implementar operaciones CRUD con vistas personalizadas.

1. **Importar los módulos necesarios**:

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

```
from rest_framework.decorators import api_view
from .models import Tarea
from .serializers import TareaSerializer
```

2. **Crear vistas para cada operación CRUD**:
    - **Listar y Crear Tareas**:

```
@api_view(['GET', 'POST'])
def tarea_list_create(request):
    if request.method == 'GET':
        tareas = Tarea.objects.all()
        serializer = TareaSerializer(tareas, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = TareaSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data, status=status.HTTP_201_CREATED)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

    - **Obtener, Actualizar y Eliminar una Tarea Específica**:

```
@api_view(['GET', 'PUT', 'DELETE'])
def tarea_detail_update_delete(request, pk):
    try:
        tarea = Tarea.objects.get(pk=pk)
    except Tarea.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)
```

```
    if request.method == 'GET':
        serializer = TareaSerializer(tarea)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = TareaSerializer(tarea, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    elif request.method == 'DELETE':
        tarea.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

Explicación de cada vista

1. **tarea_list_create**:
    - **GET**: Obtiene una lista de todas las tareas usando el serializador TareaSerializer con many=True (esto permite serializar múltiples instancias de Tarea a la vez).
    - **POST**: Crea una nueva tarea. Validamos los datos de la solicitud (request.data) y, si son válidos, guardamos la nueva tarea en la base de datos.
2. **tarea_detail_update_delete**:
    - **GET**: Obtiene los detalles de una tarea específica usando el pk (clave primaria) proporcionado en la URL.
    - **PUT**: Actualiza una tarea existente. Valida los datos de la solicitud y guarda los cambios si son válidos.
    - **DELETE**: Elimina la tarea especificada de la base de datos.

---

3. Configuración de Rutas para las Vistas CRUD

1. Abre tareas/urls.py.
2. Configura las rutas para cada vista:

```
from django.urls import path

from . import views


urlpatterns = [

    path('tareas/', views.tarea_list_create, name='tarea-list-
create'),

    path('tareas/<int:pk>/', views.tarea_detail_update_delete,
name='tarea-detail-update-delete'),

]
```

Incluir las rutas en el archivo principal de rutas urls.py del proyecto:

- Abre gestion_tareas/urls.py y añade la ruta de tareas:

```
from django.contrib import admin

from django.urls import path, include


urlpatterns = [

    path('admin/', admin.site.urls),

    path('api/', include('tareas.urls')),  # Ruta de la API de
tareas

]
```

4. Prueba de los Endpoints CRUD

Ejecuta el servidor de desarrollo para probar cada uno de los endpoints CRUD manuales:

```
python manage.py runserver
```

Endpoints Disponibles para Probar

- GET /api/tareas/: Listar todas las tareas.
- POST /api/tareas/: Crear una nueva tarea (enviar un JSON con titulo, descripcion, completada).
- GET /api/tareas/<id>/: Obtener una tarea específica por su ID.
- PUT /api/tareas/<id>/: Actualizar una tarea específica.
- DELETE /api/tareas/<id>/: Eliminar una tarea específica.

Para probar estos endpoints, puedes usar herramientas como **Postman** o el **navegador** en http://127.0.0.1:8000/api/tareas/.

Parte 3: Autenticación y Permisos en Django Rest Framework

Objetivo: Asegurar los endpoints de la API para que solo los usuarios autenticados puedan acceder a ciertas operaciones, como la creación, actualización y eliminación de tareas.

---

1. Instalación y Configuración de JWT (JSON Web Tokens)

Para agregar autenticación con JWT en DRF, usaremos el paquete **djangorestframework-simplejwt**, que permite emitir y verificar tokens JWT.

1. **Instalar el paquete djangorestframework-simplejwt**:

```
pip install djangorestframework-simplejwt
```

2. **Configurar JWT en settings.py**: Abre el archivo settings.py y añade la configuración de simplejwt en REST_FRAMEWORK:

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
'rest_framework_simplejwt.authentication.JWTAuthentication',
```

```
    ),
    'DEFAULT_PERMISSION_CLASSES': (
        'rest_framework.permissions.IsAuthenticated',
    ),
}
```

- **DEFAULT_AUTHENTICATION_CLASSES**: Define el tipo de autenticación que usará DRF; en este caso, JWTAuthentication.
- **DEFAULT_PERMISSION_CLASSES**: Establece permisos globales para todos los endpoints, permitiendo el acceso solo a usuarios autenticados.

2. Configuración de Endpoints para Obtener y Refrescar Tokens

Para que los usuarios puedan obtener y refrescar su token JWT, necesitamos configurar endpoints específicos para estos procesos.

1. Abre el archivo principal de rutas urls.py en gestion_tareas/.
2. Añade las rutas de autenticación con JWT:

```
from django.contrib import admin
from django.urls import path, include
from rest_framework_simplejwt.views import (
    TokenObtainPairView,
    TokenRefreshView,
)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('tareas.urls')),
    path('api/token/', TokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
```

```
]
```

1.
    - **TokenObtainPairView**: Endpoint para obtener un token de acceso y un token de refresco.
    - **TokenRefreshView**: Endpoint para refrescar el token de acceso utilizando el token de refresco.

Ahora, los usuarios pueden autenticarse en la API mediante estos endpoints.

---

3. Probar la Autenticación JWT

1. **Obtener un token de acceso**:
    - Realiza una solicitud POST al endpoint /api/token/ con las credenciales del usuario:

```
POST /api/token/
{
  "username": "tu_usuario",
  "password": "tu_contraseña"
}
```

La respuesta incluirá un access y un refresh token:

```
{
  "access": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1...",
  "refresh": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1..."
}
```

Usar el token de acceso para autenticar solicitudes:

- Agrega el token de acceso en los encabezados de las solicitudes que requieren autenticación:

```
Authorization: Bearer <token_de_acceso>
```

Refrescar el token de acceso:

- Realiza una solicitud POST a /api/token/refresh/ usando el token de refresco para obtener un nuevo token de acceso.

4. Configurar Permisos para los Endpoints de la API

En DRF, los permisos controlan quién puede acceder a un endpoint. Podemos definir permisos a nivel global (como en settings.py) o en cada vista individual.

Configurar permisos individuales en las vistas

1. Abre views.py en la app tareas.
2. Agrega permisos específicos a cada vista para controlar el acceso:

```
from rest_framework.permissions import IsAuthenticated
from rest_framework.decorators import api_view, permission_classes


@api_view(['GET', 'POST'])
@permission_classes([IsAuthenticated])  # Solo usuarios autenticados pueden acceder
def tarea_list_create(request):
    if request.method == 'GET':
        tareas = Tarea.objects.all()
        serializer = TareaSerializer(tareas, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
```

```
        serializer = TareaSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)


@api_view(['GET', 'PUT', 'DELETE'])
@permission_classes([IsAuthenticated])  # Solo usuarios
autenticados pueden acceder
def tarea_detail_update_delete(request, pk):
    try:
        tarea = Tarea.objects.get(pk=pk)
    except Tarea.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = TareaSerializer(tarea)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = TareaSerializer(tarea, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

```
    elif request.method == 'DELETE':
        tarea.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

En este caso, usamos el decorador @permission_classes([IsAuthenticated]) para asegurarnos de que solo los usuarios autenticados pueden acceder a estas vistas. Si un usuario no autenticado intenta acceder, recibirá un error 401 Unauthorized.

5. Opciones de Permisos en Django Rest Framework

DRF ofrece varias clases de permisos que puedes aplicar en tu API según el nivel de seguridad que necesites. Algunas opciones comunes son:

1. **AllowAny**: Permite el acceso a cualquier usuario, autenticado o no.
2. **IsAuthenticated**: Solo permite el acceso a usuarios autenticados.
3. **IsAdminUser**: Solo permite el acceso a usuarios que tienen permisos de administrador.
4. **IsAuthenticatedOrReadOnly**: Permite a los usuarios autenticados realizar cualquier operación y a los no autenticados solo operaciones de lectura (GET).

Cada una de estas opciones puede aplicarse tanto a nivel global en settings.py como en las vistas individuales, según la lógica de seguridad que se requiera.

Pruebas de Acceso y Permisos

1. **Probar Acceso Sin Token**:
    - Realiza una solicitud GET a /api/tareas/ sin incluir el encabezado Authorization.
    - Deberías recibir un error 401 Unauthorized.
2. **Probar Acceso con Token**:
    - Realiza la misma solicitud, pero esta vez incluye el encabezado Authorization con el token JWT.

- Si el token es válido, deberías obtener una respuesta 200 OK con la lista de tareas.

3. **Pruebas con diferentes permisos**:
    - Puedes experimentar con los permisos AllowAny, IsAdminUser, y otros para ver cómo cambian las respuestas dependiendo del tipo de usuario y sus permisos.

Resumen de la Autenticación y Permisos

En esta sección hemos cubierto:

- **Configuración de JWT**: Cómo instalar y configurar autenticación con JSON Web Tokens para emitir y validar tokens en DRF.
- **Rutas de autenticación**: Configurar endpoints para obtener y refrescar tokens.
- **Permisos en DRF**: Implementar permisos a nivel de vistas para controlar el acceso según el tipo de usuario.

Estos pasos aseguran que solo los usuarios autorizados pueden realizar ciertas operaciones en nuestra API, aumentando la seguridad de la aplicación.

Parte Final: Permisos y Roles Avanzados en Django Rest Framework

Objetivo: Implementar permisos personalizados y asignar roles a los usuarios para que puedan acceder o modificar diferentes recursos de la API según su rol.

---

1. Comprensión de los Permisos Personalizados en DRF

Aunque DRF ofrece permisos predeterminados como IsAuthenticated o IsAdminUser, en ocasiones es necesario crear permisos personalizados para cubrir requisitos específicos. Por ejemplo, en nuestra API de gestión de tareas, podríamos querer que solo los usuarios que crearon una tarea puedan editarla o eliminarla.

Crear Permisos Personalizados

1. Crea un archivo llamado permissions.py dentro de la app tareas.

2. Define un permiso personalizado llamado EsPropietario que permita a los usuarios modificar solo las tareas que ellos mismos crearon.

```
from rest_framework import permissions


class EsPropietario(permissions.BasePermission):
    """
    Permite que solo el creador de la tarea pueda editarla o eliminarla.
    """


    def has_object_permission(self, request, view, obj):
        # Verificar si el método es seguro (GET, HEAD, OPTIONS) o si el usuario es el propietario
        if request.method in permissions.SAFE_METHODS:
            return True
        return obj.creador == request.user
```

has_object_permission: Este método define la lógica de los permisos a nivel de objeto, permitiendo que solo los métodos seguros (GET, HEAD, OPTIONS) sean accesibles por cualquiera. Sin embargo, los métodos de modificación (PUT, DELETE) solo estarán permitidos para el propietario de la tarea.

2. Modificar el Modelo Tarea para Incluir un Campo de Propietario

Para que nuestro permiso personalizado funcione, necesitamos un campo en el modelo Tarea que almacene al usuario que creó la tarea.

1. **Abre el archivo models.py en la app tareas.**
2. **Modifica el modelo Tarea para incluir el campo creador:**

```
from django.contrib.auth.models import User

from django.db import models


class Tarea(models.Model):

   titulo = models.CharField(max_length=100)

   descripcion = models.TextField(blank=True)

   completada = models.BooleanField(default=False)

   fecha_creacion = models.DateTimeField(auto_now_add=True)

   creador = models.ForeignKey(User, on_delete=models.CASCADE)  # Propietario de la tarea


   def __str__(self):

      return self.titulo
```

Aplicar las migraciones para que el nuevo campo se incluya en la base de datos:

```
python manage.py makemigrations

python manage.py migrate
```

- **creador**: Es un campo ForeignKey que hace referencia al usuario que creó la tarea. Esto permite que cada tarea tenga un propietario específico.

3. Modificar las Vistas para Asignar el Propietario a Nuevas Tareas

Cuando un usuario crea una nueva tarea, necesitamos asignar automáticamente al usuario autenticado como el propietario.

1. Abre views.py en la app tareas.

2. Modifica la vista de creación para asignar request.user como el creador de la tarea.

```
from rest_framework import status
from rest_framework.response import Response
from rest_framework.decorators import api_view, permission_classes
from .models import Tarea
from .serializers import TareaSerializer
from .permissions import EsPropietario
from rest_framework.permissions import IsAuthenticated

@api_view(['GET', 'POST'])
@permission_classes([IsAuthenticated])
def tarea_list_create(request):
    if request.method == 'GET':
        tareas = Tarea.objects.all()
        serializer = TareaSerializer(tareas, many=True)
        return Response(serializer.data)

    elif request.method == 'POST':
        serializer = TareaSerializer(data=request.data)
        if serializer.is_valid():
            serializer.save(creador=request.user)  # Asignar el
creador
            return Response(serializer.data,
status=status.HTTP_201_CREATED)
        return Response(serializer.errors,
status=status.HTTP_400_BAD_REQUEST)
```

```
@api_view(['GET', 'PUT', 'DELETE'])
@permission_classes([IsAuthenticated, EsPropietario])  # Solo el propietario puede modificar
def tarea_detail_update_delete(request, pk):
    try:
        tarea = Tarea.objects.get(pk=pk)
    except Tarea.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = TareaSerializer(tarea)
        return Response(serializer.data)

    elif request.method == 'PUT':
        serializer = TareaSerializer(tarea, data=request.data)
        if serializer.is_valid():
            serializer.save()
            return Response(serializer.data)
        return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)

    elif request.method == 'DELETE':
        tarea.delete()
        return Response(status=status.HTTP_204_NO_CONTENT)
```

- **serializer.save(creador=request.user)**: Al crear una nueva tarea, asignamos automáticamente al usuario autenticado como el creador.

---

4. Prueba de los Permisos de Propietario

Ejecuta el servidor y realiza las siguientes pruebas para verificar que el sistema de permisos está funcionando como se espera:

1. **Crear una Tarea como Usuario Autenticado**:
    - Obtén un token de autenticación y crea una tarea usando ese token.
    - Confirma que el creador de la tarea es el usuario autenticado.
2. **Intentar Editar o Eliminar Tareas de Otro Usuario**:
    - Cambia el token para simular que otro usuario intenta acceder a la tarea.
    - Realiza una solicitud PUT o DELETE en la tarea creada por el primer usuario y verifica que recibe un error 403 Forbidden.
3. **Acceso de Solo Lectura para Otros Usuarios**:
    - Verifica que otros usuarios aún pueden realizar operaciones de lectura (GET) en las tareas, pero no pueden modificarlas ni eliminarlas.

Roles Adicionales Usando Permisos de Django

Si se requieren **roles adicionales**, como un rol de administrador, puedes hacer uso de los permisos de usuario que Django proporciona. Algunos ejemplos son:

1. **Verificar si un usuario es is_staff**:
    - Solo permitir el acceso a ciertas vistas para usuarios con el rol de staff:

```
@permission_classes([IsAdminUser])  # Permite solo a
administradores
```

1. **Permiso IsAuthenticatedOrReadOnly**:
    - Permite el acceso completo a los usuarios autenticados, pero limita a los no autenticados a operaciones de lectura.

Resumen Final

En esta última sección hemos aprendido:

- Cómo crear **permisos personalizados** en DRF para controlar el acceso a nivel de objeto.
- Modificar el modelo para **asignar un propietario** a cada tarea, permitiendo que solo el creador la modifique o elimine.
- Configurar los permisos en las vistas para que el acceso a la API sea más seguro y controlado.

Esta estructura permite crear una API segura y personalizada, donde cada usuario puede gestionar únicamente sus propias tareas, y con opciones para roles adicionales en caso de que la aplicación lo necesite.