Manual de desarrollo: Extensión del backend Django y DRF

Este documento guía a futuros desarrolladores sobre cómo añadir nuevas tablas (modelos) al sistema de base de datos, y cómo exponerlas mediante operaciones CRUD a través de la API interna utilizando Django y Django REST Framework (DRF).

1. Estructura del Proyecto

A continuación, se presenta la estructura de directorios y archivos del backend del proyecto, basada en el uso de Django. La estructura está organizada para facilitar el mantenimiento y la escalabilidad del sistema. Cada componente del sistema tiene su propia carpeta o archivo, lo que facilita la organización del código y la separación de responsabilidades[1].

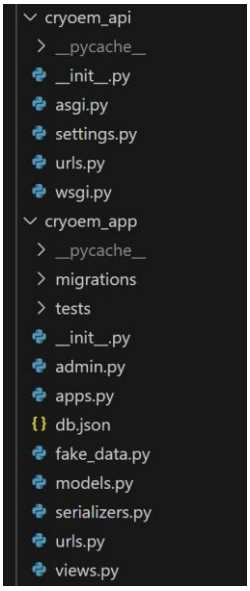

Figura 1. Estructura de las carpetas del proyecto.

El backend está estructurado en los siguientes componentes clave:

- **models.py:** Define la estructura de la base de datos mediante clases de Python. Cada clase representa una tabla. Los atributos representan

[1] La siguiente imagen muestra una captura de pantalla de la estructura de carpetas del proyecto en Visual Studio, donde se pueden observar los directorios y archivos principales.

columnas, y se pueden definir relaciones (ForeignKey, OneToOne, etc.). Ejemplo: Micrograph, CTF, Particle.

- **serializers.py:** Serializa/deserializa los modelos para la API. Django almacena datos como objetos complejos (*QuerySets*). Para enviarlos al frontend (React), se necesitan convertir esos objetos en JSON. El *serializer* traduce los objetos de Django a Python (diccionarios) y luego a JSON. También convierte datos JSON recibidos del frontend en objetos válidos para guardar en la base de datos.
- **views.py:** Contiene la lógica de negocio y las vistas (o controladores) que gestionan las operaciones CRUD (crear, leer, actualizar, eliminar). Puede incluir lógica personalizada según las necesidades de la aplicación.
- **urls.py** (en cryoem_app): Define las rutas de los *endpoints*. Define qué rutas existen en la API y a qué vista apuntan. Esto incluye:
    - Rutas automáticas para CRUD *(router.register())*.
    - Rutas manuales para vistas específicas *(path('api/config/', ...))*.
- **admin.py:** Registra modelos para que puedan ser gestionados visualmente en el panel admin (en la URL: http://127.0.0.1:8000/admin/). Esto es útil para edición manual o verificación rápida de datos en desarrollo.

2. Crear un Nuevo Modelo

Para añadir una nueva tabla a la base de datos:

1. Abre models.py.
2. Define una nueva clase que herede de *models.Model*.
3. Declara los campos necesarios utilizando tipos de Django ORM (*CharField, FloatField*, etc.).
4. Añade validaciones si es necesario (*MinValueValidator, MaxLength, etc.)*.
5. Opcional: implementa *__str__()* para facilitar la lectura en paneles y consola. Permite mostrar nombres legibles en el panel de administración de Django y logs.

Por ejemplo:

```python
from django.db import models
from django.core.validators import MinValueValidator

class SampleModel(models.Model):
    name = models.CharField(max_length=100)
    value = models.FloatField(validators=[MinValueValidator(0.0)])

    def __str__(self):
        return f"{self.name} ({self.value})"
```

Figura 2. Ejemplo de creación de un modelo.

3. Crear un Serializador

1. Abre serializers.py.
2. Importa el nuevo modelo.
3. Crea una clase que herede de *serializers.ModelSerializer*.
4. Define validaciones personalizadas si aplica.

Por ejemplo:

```python
from rest_framework import serializers
from .models import SampleModel

class SampleModelSerializer(serializers.ModelSerializer):
    value = serializers.FloatField(min_value=0.01)

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

Figura 3. Ejemplo de creación de un Seralizador.

4. Crear la Vista (View)

Opción A: *ViewSet* para CRUD completo

1. Abre views.py.
2. Importa el modelo y su *serializer*.

3. Crea una clase que herede de *viewsets.ModelViewSet*.

Por ejemplo:

```python
from rest_framework import viewsets
from .models import SampleModel
from .serializers import SampleModelSerializer

class SampleModelViewSet(viewsets.ModelViewSet):
    queryset = SampleModel.objects.all()
    serializer_class = SampleModelSerializer
```

Figura 4. Ejemplo de creación de una Vista.

Opción B: APIView para lógica personalizada

Usa APIView cuando necesites más control (GET personalizado, PATCH parcial, etc.).

5. Registrar el Endpoint en URLs

1. Abre urls.py.
2. Usa un router DRF para registrar la vista si es un *ViewSet*.
3. Para APIView, añade la ruta manualmente.

Ejemplo (con router):

```python
from rest_framework import routers
from .views import SampleModelViewSet

router = routers.DefaultRouter()
router.register(r'samplemodel', SampleModelViewSet)

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

Figura 5. Ejemplo de registro de un punto de acceso en URLs (con router).

Ejemplo (sin router, usando APIView):

```python
from .views import SampleModelAPIView

urlpatterns += [
    path('api/samplemodel/', SampleModelAPIView.as_view(), name='samplemodel'),
]
```

Figura 6. Ejemplo de registro de un punto de acceso en URLs (sin router).

6. Registrar el modelo en el Administrador

Esto permite editar datos manualmente desde el panel de administración de Django.

1. Abre admin.py.
2. Importa y registra el modelo.

Ejemplo:

```python
from django.contrib import admin
from .models import SampleModel

admin.site.register(SampleModel)
```

Figura 7. Ejemplo de registro del modelo en el panel de administrador.

7. Migraciones

Después de añadir/modificar modelos, se debe aplicar los cambios al esquema de la base de datos, mediante los siguientes comandos:

```
python manage.py makemigrations
```

```
python manage.py migrate
```

8. Probar con datos simulados de *CTF, Micrograph y Config*

Para visualizar los datos simulados, se deben ejecutar los siguientes comandos en la terminal:

Primero, si se ha realizado en el archivo fake_data.py, debes ejecutar el siguiente comando para abrir el *shell* de Django y cargar los datos simulados definidos en el archivo *fake_data.py*:

```
python manage.py shell < cryoem_app/fake_data.py
```

Después, mediante el siguiente comando, usar el script load_data.py, que insertará los datos simulados en la base de datos, permitiendo que la aplicación los muestre y los utilice para las pruebas:

```
python load_data.py
```

9. Probar Endpoints de la API con autenticación por token

Para interactuar con los endpoints de la API, es necesario autenticarte utilizando un token de autenticación. Este token se obtiene a través de la página de inicio de sesión de la aplicación, y debe ser incluido en las peticiones a la API para poder acceder a los recursos protegidos:

1. Inicia sesión en la aplicación web:
    a. Ve a la página de inicio de sesión: http://localhost:8000/login.
    b. Introduce tus credenciales (usuario y contraseña).
    c. Al iniciar sesión con éxito, el sistema generará un token de autenticación para ti.

d. Nota: El token de autenticación se mostrará en la respuesta de la API cuando inicies sesión, y este es el token que usarás en tus solicitudes. Si no se muestra, puedes obtenerlo en las herramientas de desarrollo del navegador o revisando la respuesta en la consola.
e. También puedes abrir http://localhost:8000/docs/ para ver la documentación interactiva.

Obtener token con Postman

Puedes obtener tu token haciendo una solicitud POST a:

`http://127.0.0.1:8000/api/auth/token/`

En el cuerpo de la solicitud (Body → raw → JSON) debes incluir:

```
{
  "username": "tu_usuario",
  "password": "tu_contraseña"
}
```

Si las credenciales son correctas, recibirás una respuesta como esta:

```
{
  "token": "tu_token_de_autenticación"
}
```

Usa ese token en futuras solicitudes añadiendo este encabezado (Headers):

Authorization: Token tu_token_de_autenticación

Usar el token en Postman

Una vez que hayas obtenido el token de autenticación desde el endpoint correspondiente (/api/auth/token/), debes incluirlo en las cabeceras de tus peticiones para poder acceder a los endpoints protegidos de la API.

Después, para usar el token en Postman:

1. Abre Postman y crea una nueva petición.
2. Ve a la pestaña Headers.
3. Añade un nuevo campo con: Key: Authorization Value: Token TU_TOKEN_AQUI (Reemplaza TU_TOKEN_AQUI por el valor real obtenido del endpoint de login, por ejemplo: Token 92d8b79030ad215d57804e801bhjdf893h88o0).
4. Envía la petición normalmente. Si el token es válido, recibirás una respuesta autorizada (200 OK); si no, una 401 Unauthorized.
5. Ahora puedes probar los endpoints de la API, añadiendo el token a las cabeceras de cada solicitud:
    - POST /api/samplemodel/ → Crear
    - GET /api/samplemodel/ → Listar
    - GET /api/samplemodel/{id}/ → Obtener uno concreto
    - PATCH /api/samplemodel/{id}/ → Actualizar parcial
    - DELETE /api/samplemodel/{id}/ → Eliminar

10. Buenas Prácticas

- Usa validaciones en los serializadores para evitar datos incorrectos.
- Agrega *__str__()* en modelos para facilitar el debugging.
- Reutiliza lógica común creando mixins o clases base, especialmente cuando la aplicación comience a escalar o compartir comportamientos similares entre vistas o serializadores.
- Mantén nombres de rutas coherentes y en minúsculas.
- Versiona la API si hay cambios mayores (/api/v1/, etc.).

11. Notas finales

Documentación automática: La API del proyecto genera documentación automáticamente utilizando Django REST Framework (DRF) y CoreAPI. Esta documentación es accesible de manera interactiva a través de la siguiente URL:

http://localhost:8000/docs/

La interfaz generada permite a los desarrolladores explorar y probar los diferentes endpoints disponibles en la API. Al acceder a esta URL, se podrá visualizar:

- Métodos HTTP disponibles: como GET, POST, PATCH, DELETE.
- Estructura de los datos: los campos esperados en las solicitudes y las respuestas, incluyendo tipos de datos y validaciones.
- Ejemplos de uso: cómo interactuar con cada endpoint y qué parámetros enviar.

Es importante tener en cuenta que la documentación interactiva solo está disponible cuando la aplicación backend está en ejecución. Para ello, se debe levantar el servidor de desarrollo de Django con el siguiente comando:

python manage.py runserver