# Curso Django

## 1. Creación del entorno virtual e instalación de Django

Lo primero de todo es crear un entorno virtual en nuestra carpeta de trabajo. En este caso vamos a llamar a nuestra carpeta de trabajo “django-crash-course”.

Al estar trabajando con Python, el comando para crear un nuevo entorno de trabajo es: python -m venv {venv} - {venv} –> Lo sustituimos con el nombre que queramos. En mi caso **.env**

### 1.1. Activación del entorno virtual

Para la instalación y gestión de los paquetes de nuestro proyecto debemos activar nuestro entorno virtual. Dependiendo de nuestro SO, el comando puede variar: > CMD venv\Scripts\activate.bat

Poweshell venv\Scripts\Activate.ps1

Git Bash source venv/Scripts/activate

Nosotros estamos trabajando en Windows, con una terminal CMD. Por lo que nuestro comando es: .env\Scripts\activate

Si lo hemos hecho bien, nuestra terminal deberá aparecer de la siguiente manera: > (.env) PS D:-crash-course>

### 1.2. Instalación de Django

Una vez que tenemos activado nuestro entorno virtual, podemos instalar las dependencias necesarias en nuestro proyecto. Django no viene instalado por defceto en Python ,por lo que tendremos que descargarlo e instalarlo usando el gestor de paquetes **pip**

En la terminal lanzamos el siguiente comando: pip install django

Automáticamente, Django y todas sus dependencias se descargarán y se isntalarán en nuestro entorno virtual y ya podremos empezar a usar Django :)

## 2. Primeros pasos con Django

### 2.1. Creación de un proyecto en Django

Una vez que hemos instalado Django en nuestro entorno virtual, podemos crear nuestro primer proyecto. Podemos comprobar que Django se ha instalado correctamente deesta manera: python -m django --version

El siguiente paso es crear nuestro proyecto, directamente desde la terminal: django-admin startproject mysite

Ester comando lo que hará será crear un proyecto de Django llamado **mysite**. > Debes evitar nombrar proyectos con nombres de componentes integrados de Python o Django. Esto significa que debes evitar usar nombres como django(que entrará en conflicto con el propio Django) o test(que entra en conflicto con un paquete integrado de Python).

En mi caso, a la misma altura que el .env, he creado una carpeta nueva llamada **“django-crash-course”** y después he utilizado el siguiente comando: django-admin startproject mysite django-crash-course

De esta manera, se ha creado mi proyecto de Django de nombre **mysite**, dentro de la carpeta **“django-crash-course”**.

Automáticamente se habrá creado mi estructura principal del proyecto: - djangotutorial/ - manage.py - mysite/ - **init**.py - settings.py - urls.py - asgi.py - wsgi.py

Estos archivos son:

* manage.py –> Una utilidad de línea de comandos que te permite interactuar con este proyecto de Django de diversas maneras. Puedes leer todos los detalles manage.py en django-admin y manage.py .
* mysite/ –> Un directorio que contiene el paquete de Python de tu proyecto. Su nombre es el nombre del paquete de Python que necesitarás para importar cualquier contenido (por ejemplo, mysite.urls).
* mysite/**init**.py –> Un archivo vacío que indica a Python que este directorio debe considerarse un paquete. Si eres principiante en Python, lee más sobre paquetes en la documentación oficial de Python.
* mysite/settings.py –> Configuración de este proyecto de Django. La configuración de Django te explicará todo sobre su funcionamiento.
* mysite/urls.py –> Las declaraciones de URL de este proyecto de Django; una tabla de contenido de tu sitio web basado en Django.
* mysite/asgi.py –> Un punto de entrada para servidores web compatibles con ASGI que atiendan tu proyecto.
* mysite/wsgi.py –> Un punto de entrada para servidores web compatibles con WSGI que atiendan tu proyecto.

Ya tendríamos Django listo, y si ejecutamos dentro de nuestra nueva carpeta el siguiente comando, podremos comprobar que nuestra aplicación ya está corriendo: python manage.py runserver

Por defecto, como buen backend, esto se ejecuta en la ruta http://127.0.0.1:8000/; pero podemos cambiar el puerto de esta manera: python manage.py runserver 8001

Así, podemos especificar el puerto en el que correr nuestra aplicación.

### 2.2 . Proyectos vs. Apps y tu primera App

Perfecto, ya has creado el proyecto mysite. Ahora toca entender uno de los conceptos más importantes de Django: la diferencia entre un Proyecto y una App.

* **Proyecto Django**: Piensa en el proyecto como el contenedor de todo tu sitio web. Gestiona la configuración global (settings.py), las rutas URL principales (urls.py) y agrupa una o más “apps”.
* **App Django**: Una “app” es un módulo autocontenido que realiza una función específica. Por ejemplo, podrías tener una app para los usuarios, otra para los productos, y otra para los pedidos. Las apps están diseñadas para ser reutilizables.

Nuestro objetivo es crear una API, así que vamos a crear una app dedicada a gestionar toda la lógica de la API.

#### Paso 1: Crear nuestra primera app

1. Abre tu terminal.
2. Asegúrate de que estás dentro de la carpeta de tu proyecto, al mismo nivel que el archivo manage.py.
3. Ejecuta el siguiente comando para crear una app llamada **api**: python manage.py startapp api

Ahora, si miras tu estructura de carpetas, verás que ha aparecido un nuevo directorio llamado api con varios archivos dentro (models.py, views.py, etc.). ¡Esa es tu primera app!

#### Paso 2: Registrar la App en el proyecto

Crear la app no es suficiente; debemos decirle a nuestro proyecto mysite que esta nueva app api existe y que debe tenerla en cuenta.

1. Abre el archivo mysite/settings.py.
2. Busca la lista llamada **INSTALLED\_APPS**.
3. Añade el nombre de tu app (‘api’) al final de la lista. Te recomiendo añadir un comentario para saber cuáles son tus apps locales.
4. Guarda el archivo.

A mi personalmente me gusta diferenciar dentro de esta lista que apps son “base”, cuales son mías, y cuales son de terceros. Las veremos más adelante. De maenra que se vería así:

# mysite/settings.py  
INSTALLED\_APPS = [  
 'django.contrib.admin',  
 'django.contrib.auth',  
 'django.contrib.contenttypes',  
 'django.contrib.sessions',  
 'django.contrib.messages',  
 'django.contrib.staticfiles',  
  
 # My Apps  
 'api',  
  
 # Third Apps  
]

¡Felicidades! Has creado y registrado correctamente tu primera aplicación. Este es el flujo de trabajo fundamental para añadir nuevas funcionalidades a cualquier proyecto de Django.

### 2.3. Tu Primer Modelo y las Migraciones

En Django, un Modelo es la única y definitiva fuente de información sobre tus datos. Es una clase de Python que representa una tabla en tu base de datos. Django usa estos modelos para crear las tablas y para interactuar con ellas (leer, escribir, actualizar, etc.).

Para nuestro proyecto, vamos a crear un modelo simple pero muy común: una Tarea (Task).

#### Paso 1: Creando el Modelo “Task”

1. Abre el archivo api/models.py. Por defecto, estará casi vacío.
2. Reemplaza su contenido con el siguiente código para definir nuestro modelo Task:

* # api/models.py from django.db import models
* class Task(models.Model): title = models.CharField(max\_length=200) description = models.TextField(blank=True, null=True) completed = models.BooleanField(default=False) created\_at = models.DateTimeField(auto\_now\_add=True)
* def \_\_str\_\_(self):  
   return self.title
* **class Task(models.Model)** –> Define una clase llamada Task que hereda de **models.Model**. Así es como Django sabe que esto es un modelo.
* **title = models.CharField(…)** –> Un campo de texto para el título. max\_length=200 es obligatorio para los CharField.
* **description = models.TextField(…)** –> Un campo para textos largos. blank=True, null=True significa que este campo es opcional.
* **completed = models.BooleanField(…)**: Un campo que será True o False. Por defecto (default=False), las tareas se crearán como no completadas.
* **created\_at = models.DateTimeField(…)**: Un campo de fecha y hora. auto\_now\_add=True le dice a Django que guarde la fecha y hora actual automáticamente cuando se cree una nueva tarea.
* **def str(self):**: Este método especial le dice a Django cómo “mostrar” un objeto **Task** de forma legible (por ejemplo, en el panel de administrador). Aquí, le decimos que muestre el título de la tarea.

#### Paso 2: Las Migraciones (el ‘Control de Versiones’ de tu Base de Datos)

Ahora que hemos definido el modelo en Python, necesitamos traducir esa definición a un formato que la base de datos entienda (lenguaje SQL) y aplicarlo. Este proceso se hace en dos pasos.

1. **Crear el archivo de migración**: Este comando analiza tus models.py, detecta los cambios (en este caso, la creación del modelo Task) y genera un archivo de instrucciones en la carpeta api/migrations/.

Abre tu terminal (asegúrate de estar en la carpeta del proyecto con manage.py) y ejecuta: python manage.py makemigrations

Verás una salida parecida a: Migrations for ‘api’: api/migrations/0001\_initial.py.

**Aplicar la migración a la base de datos**: Este comando toma todos los archivos de migración que no se han aplicado y los ejecuta contra la base de datos, creando las tablas y columnas necesarias.

Ejecuta el siguiente comando: python manage.py migrate Django aplicará no solo tu migración, sino también las migraciones iniciales de las otras apps que vienen por defecto (admin, auth, etc.).

¡Excelente! Has definido la estructura de tus datos y has creado la tabla correspondiente en la base de datos sin escribir una sola línea de SQL.

Y te preguntarás, ¿Cómo puede funcionar esto si no he creado ninguna base de datos, ni ninguna conexión, no he escrito SQL…? Muy sencillo.

En Django no necesitas crear manualmente la base de datos ni escribir SQL al principio porque Django está diseñado para abstraer y automatizar esas tareas usando su sistema de ORM (Object-Relational Mapping). Te explico por qué paso a paso:

Cuando creas un proyecto Django, en **settings.py** ya tienes configurada una base de datos por defecto:

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.sqlite3',  
 'NAME': BASE\_DIR / "db.sqlite3",  
 }  
}

Eso significa que Django usará SQLite por defecto, que es un motor de base de datos ligero basado en archivos. Además, no necesitas instalar ni configurar un servidor de base de datos (como PostgreSQL o MySQL) al principio. Cuando ejecutas python manage.py migrate, Django crea el archivo db.sqlite3 automáticamente.

Ahora mismo estamos trabajando en un proyecto muy pequeño, por lo que no es necesario configurar otro servidor de base de datos, pero lo suyo es usar PostsgreSQL o MySQL, más grandes y potentes.

Te dejo por aquí la forma de configurar una base de datos más grande por si te interesa: #### Configuración de PostrgeSQL en lugar de SQLite - Instala el conector de PostgreSQL Primero, necesitas instalar el paquete que permite a Django conectarse con PostgreSQL: pip install psycopg2-binary - Crea la base de datos en PostgreSQL Abre psql o tu gestor favorito y ejecuta:

CREATE DATABASE midb;  
CREATE USER miusuario WITH PASSWORD 'miclave';  
ALTER ROLE miusuario SET client\_encoding TO 'utf8';  
ALTER ROLE miusuario SET default\_transaction\_isolation TO 'read committed';  
ALTER ROLE miusuario SET timezone TO 'UTC';  
GRANT ALL PRIVILEGES ON DATABASE midb TO miusuario;

* Configura settings.py de Django Abre el archivo settings.py de tu proyecto y busca esta sección:
* DATABASES = { ‘default’: { ‘ENGINE’: ‘django.db.backends.sqlite3’, ‘NAME’: BASE\_DIR / “db.sqlite3”, } }

Reemplázala por:

DATABASES = {  
 'default': {  
 'ENGINE': 'django.db.backends.postgresql',  
 'NAME': 'midb',  
 'USER': 'miusuario',  
 'PASSWORD': 'miclave',  
 'HOST': 'localhost',  
 'PORT': '5432',  
 }  
}

* Ejecuta migraciones Una vez configurado, ejecuta los siguientes comandos: python manage.py makemigrations python manage.py migrate Esto creará todas las tablas en la base de datos PostgreSQL que configuraste.
* Verifica que funciona python manage.py runserver ### 2.4. Panel de Administrador de Django

Una de las características más potentes de Django es su panel de administración automático. Es una interfaz web profesional y lista para usar que te permite a ti (como administrador del sitio) gestionar el contenido de tu aplicación. Vamos a hacer que nuestro modelo Task aparezca en este panel.

#### Paso 1: Resgistrar el modelo Task en el Admin

Para que un modelo sea visible y gestionable en el panel de administrador, tienes que registrarlo explícitamente. 1. Abre el archivo **api/admin.py** 2. Importa tu modelo Task y regístralo usando **admin.site.register():**

# api/admin.py  
from django.contrib import admin  
from .models import Task # Importamos el modelo Task  
  
admin.site.register(Task)

Con estas dos líneas de código, le has dicho a Django que cree una interfaz completa para gestionar tareas.

#### Paso 2: Crear un superusuario

Para poder acceder al panel de administrador, necesitas una cuenta de usuario con permisos de administrador. - En tu terminal ejecuta el siguiente comando: python manage.py createsuperuser

* El sistema te pedira que introduzcas: Nombre de usuario: Elige uno (ej: admin). Dirección de correo electrónico: Puedes poner una real o una de prueba. Contraseña: Escribe una contraseña. Por seguridad, no verás los caracteres mientras escribes. Confirmar contraseña: Vuelve a escribirla.

Si todo ha ido bien, verás un mensaje de “Superuser created successfully.”

#### Paso 3: Explorar el Panel de Administración de Django

Esta es la parte divertida y donde realmente puedes empezar a ver el potencial de Django, su escalabilidad, su robustez y el control que le da al desarrollador. - Levanta el servidor de desarrollo (si no lo tenías ya funcionando): python manage.py runserver - Abre tu navegador web y ve a la siguiente URL: http://127.0.0.1:8000/admin/ - Verás la página de inicio de sesión del administrador de Django. Usa el nombre de usuario y la contraseña que acabas de crear.

¡Ya estás dentro! Verás una sección llamada “API” con un enlace a “Tasks”.

Desde aquí, podremos directamente crear nuevas tasks, eliminarlas, editarlas, etc… **Haz clic en “Tasks” y luego en el botón “Add task +” de la esquina superior derecha.**

Puedes crear tu primera tarea usando el formulario que Django ha generado automáticamente a partir de tu modelo. Rellena el título, la descripción (si quieres) y verás cómo los campos *completed* y *created\_at* se comportan como definimos. Esto es ideal para administrar tu aplicación sin necesidad de crear interfaces de gestión desde cero.

Además, nosotros podemos personalizar nuestra interfaz de adminsitración para mostrar los datos que más nos interesan. Por ejemplo, podemos especificar que campos del modelo se muestren, los campos de búsqueda y los filtros que podemos aplicar. Vamos a mostrar en el listado de tareas los campos “title”, “completed” y “created\_at”, vamos a decirle al panel de adminsitración que nos permita buscar las tareas por su campo “title” y vamos a decirle que queremos tener la opción de filtrar por tareas completadas.

Abre tu archivo **api/admin.py** y modifícalo para que se vea así:

# api/admin.py  
from django.contrib import admin  
from .models import Task  
  
class TaskAdmin(admin.ModelAdmin):  
 """  
 Personaliza la vista de lista y el formulario de edición para el modelo Task.  
 """  
 list\_display = ('title', 'completed', 'created\_at')  
 search\_fields = ('title',)  
 list\_filter = ('completed',)  
  
# Registra el modelo Task con la clase de personalización TaskAdmin  
admin.site.register(Task, TaskAdmin)

**¿Qué Has Hecho Exactamente?** Al crear la clase **TaskAdmin** que hereda de **admin.ModelAdmin**, le estás diciendo a Django: “Oye, para el modelo Task, no uses la vista de administrador por defecto, usa esta configuración personalizada”.

* list\_display > Muestra estas columnas en la lista de tareas. Es mucho más útil que ver solo el título.
* search\_fields > Añade una barra de búsqueda en la parte superior que buscará por el campo title.
* list\_filter > Añade una barra lateral para filtrar rápidamente las tareas (por ejemplo, ver solo las completadas o las pendientes).

#### Alternativa con decoradores

Una forma más moderna y “Pythonica” de hacer exactamente lo mismo es usando un decorador. Es más conciso y muchos desarrolladores lo prefieren.

# api/admin.py  
from django.contrib import admin  
from .models import Task  
  
@admin.register(Task)  
class TaskAdmin(admin.ModelAdmin):  
 list\_display = ('title', 'completed', 'created\_at')  
 search\_fields = ('title',)  
 list\_filter = ('completed',)

Ambas formas son 100% correctas y funcionales. Elige la que más te guste.

Ahora, si vuelves a ejecutar tu servidor (python manage.py runserver) y visitas la página de administración de Tareas, verás una interfaz mucho más rica y útil.

### 2.5. Tu Primera API con Django REST Framework

Django REST Framework (DRF) es un kit de herramientas que se integra con Django para construir APIs web de forma rápida y flexible. Se encarga del trabajo pesado de convertir tus datos a formato JSON, gestionar la autenticación, los permisos y mucho más.

Nuestro objetivo: crear un endpoint (una URL de la API) en /api/tasks/ que devuelva una lista de todas nuestras tareas.

#### Paso 1: Instalar y configurar DRF

* Detén tu servidor (si está funcionando) y, en la terminal, instala la librería: pip install djangorestframework
* Ahora, al igual que hicimos con nuestra app api, debemos registrar DRF en nuestro proyecto. Abre mysite/settings.py y añade ‘rest\_framework’ a la lista de INSTALLED\_APPS.
* # mysite/settings.py  
   INSTALLED\_APPS = [  
   # ... otras apps  
   'api',  
    
   # Apps de terceros  
   'rest\_framework',  
   ]

#### Paso 2: El Serializer (El traductor de datos)

Un Serializer en DRF traduce datos complejos, como los objetos de nuestro modelo Task, a un formato que se puede enviar fácilmente por internet, como JSON. También hace el trabajo inverso: valida y convierte datos JSON en objetos de Django.

* Crea un nuevo archivo en tu app: api/serializers.py.
* Añade el siguiente código: # api/serializers.py from rest\_framework import serializers from .models import Task
* class TaskSerializer(serializers.ModelSerializer):  
   class Meta:  
   model = Task  
   fields = '\_\_all\_\_' # Incluye todos los campos del modelo

Con ModelSerializer, DRF crea automáticamente un serializador con los campos que coinciden con los de nuestro modelo Task. ¡Es así de simple!

#### Paso 3: La Vista (La lógica de la API)

La vista se encarga de recibir una petición web y devolver una respuesta. Usaremos una de las “vistas genéricas” de DRF que nos ahorra escribir código repetitivo.

* Modifica tu archivo api/views.py: # api/views.py from rest\_framework import generics from .models import Task from .serializers import TaskSerializer
* class TaskViewSet(generics.ListAPIView):  
   queryset = Task.objects.all()  
   serializer\_class = TaskSerializer
* **generics.ListAPIView** > Es una vista pre-construida por DRF para manejar peticiones que listan un conjunto de objetos.
* **queryset** > Le dice a la vista qué objetos debe obtener de la base de datos (todos los objetos Task).
* **serializer\_class** > Le dice a la vista qué serializador debe usar para traducir esos objetos.

#### Paso 4: La URL (La puerta de entrada)

Finalmente, necesitamos conectar una URL a la vista que acabamos de crear. - Crea un archivo urls.py dentro de tu app api. Este es el lugar correcto para las URLs específicas de la API. # api/urls.py from django.urls import path from .views import TaskViewSet

urlpatterns = [  
 path('tasks/', TaskViewSet.as\_view(), name='task-list'),  
 ]

* Ahora, dile al proyecto principal que tenga en cuenta las URLs de tu app api. Modifica el archivo principal mysite/urls.py:
* # mysite/urls.py  
   from django.contrib import admin  
   from django.urls import path, include # ¡Asegúrate de que 'include' esté importado!  
    
   urlpatterns = [  
   path('admin/', admin.site.urls),  
   path('api/', include('api.urls')), # Incluye las URLs de la app 'api'  
   ]

#### Paso 5: Prueba tu API

* Arranca el servidor: python manage.py runserver
* Abre tu navegador y ve a: http://127.0.0.1:8000/api/tasks/

Verás la Browsable API de DRF, una interfaz web que te muestra tus datos en formato JSON y te permite interactuar con tu API directamente desde el navegador. ¡Deberías ver las tareas que creaste en el panel de administrador!

¡Lo has conseguido! Has creado tu primer endpoint de solo lectura. Una aplicación externa ya podría “leer” tus tareas.

### 2.6. Crear tareas a través de la API

Ahora que ya podemos leer la lista de tareas (GET), el siguiente paso es poder crear nuevas tareas enviando datos a la API (peticiones POST). Gracias a Django REST Framework, esto es increíblemente sencillo.

#### Paso 1: Actualizar la Vista

Solo tenemos que hacer un pequeño cambio en api/views.py. Vamos a cambiar la vista ListAPIView (que solo permite listar) por ListCreateAPIView (que permite listar y crear). - Abre tu archivo api/views.py. - Modifica la clase de la que hereda tu vista: # api/views.py from rest\_framework import generics from .models import Task from .serializers import TaskSerializer

# Cambia ListAPIView por ListCreateAPIView  
 class TaskViewSet(generics.ListCreateAPIView):  
 queryset = Task.objects.all()  
 serializer\_class = TaskSerializer

#### Paso 2: Probar el endpoint

* Asegúrate de que el servidor está corriendo.
* Refresca la página en tu navegador: http://127.0.0.1:8000/api/tasks/

Ahora verás algo nuevo y muy potente: en la parte inferior de la API Navegable, aparecerá un formulario HTML. Puedes usar ese formulario para rellenar los datos de una nueva tarea y enviarla con un POST.

¡Pruébalo! Crea una nueva tarea desde el navegador. Verás cómo, tras enviarla, la página se recarga y tu nueva tarea aparece en la lista JSON de arriba.

¡Felicidades, tu API ahora puede leer y escribir datos!

En el desarrollo de APIs, esto se conoce como CRUD (Create, Read, Update, Delete). Ya tenemos la “C” (Crear) y la “R” (Leer la lista). Ahora vamos a por la “U” (Actualizar) y la “D” (Borrar), además de leer un solo elemento.

### 2.7. Detalle, Actualización y Borrado (CRUD completo)

Para poder ver, modificar o borrar una tarea específica, necesitamos un nuevo “endpoint” que identifique esa tarea, como /api/tasks/1/. A esto se le llama una vista de detalle.

DRF nos lo pone muy fácil con otra vista genérica.

#### Paso 1: Crear la Vista de Detalle

Vamos a añadir una nueva vista en api/views.py. Esta única vista se encargará de recuperar (GET), actualizar (PUT/PATCH) y eliminar (DELETE) un objeto individual. - Abre api/views.py. - Añade esta nueva clase: # api/views.py from rest\_framework import generics from .models import Task from .serializers import TaskSerializer

class TaskViewSet(generics.ListCreateAPIView):  
 queryset = Task.objects.all()  
 serializer\_class = TaskSerializer  
  
 # NUEVA VISTA  
 class TaskRetrieveUpdateDestroyAPIView(generics.RetrieveUpdateDestroyAPIView):  
 queryset = Task.objects.all()  
 serializer\_class = TaskSerializer

Como ves, la configuración es idéntica. DRF se encarga de la lógica interna para manejar un solo objeto en lugar de una lista.

#### Paso 2: Añadir la Nueva URL

Ahora necesitamos conectar esta nueva vista a una URL que pueda capturar el ID de la tarea. - Abre api/urls.py. - Añade un nuevo path a la lista urlpatterns: # api/urls.py from django.urls import path from .views import TaskListCreateAPIView, TaskRetrieveUpdateDestroyAPIView # Importa la nueva vista

urlpatterns = [  
 path('tasks/', TaskListCreateAPIView.as\_view(), name='task-list-create'),  
 # NUEVA RUTA  
 path('tasks/<int:pk>/', TaskRetrieveUpdateDestroyAPIView.as\_view(), name='task- detail'),  
 ]

La parte clave aquí es . Esto le dice a Django que espere un número entero en esa parte de la URL y que se lo pase a la vista como un argumento llamado pk (de Primary Key). Así es como la vista sabe qué tarea específica tiene que buscar.

#### Paso 3: Probar la API Completa

* Asegúrate de que tu servidor está corriendo python manage.py runserver
* Ve en tu navegador a la URL de una de las tareas que ya has creado, por ejemplo: http://127.0.0.1:8000/api/tasks/1/

Ahora verás la página de la API Navegable para esa tarea individual. Desde ahí puedes:

* Ver sus datos en formato JSON.
* Usar el formulario para modificarla (PUT).
* Hacer clic en el botón “DELETE” para borrarla.

¡Felicidades! Has construido una API REST completamente funcional con todas las operaciones CRUD. Has sentado la base fundamental sobre la que se construyen todas las APIs complejas.

Sin embargo, este tan solo es un ejemplo creado para que podamos entender como funciona un CRUD en una API REST. Si queremos seguir buenas prácticas, hacerlo de una forma más profesional y eficiente podemos hacerlo de otra manera.

### 2.8. Refactorizando a ViewSets y Routers (+ profesional)

Un ViewSet es una clase que agrupa toda la lógica de un recurso (en nuestro caso, las Tareas). Usaremos un ModelViewSet, que nos da toda la funcionalidad CRUD (list, create, retrieve, update, destroy) de forma gratuita.

Esto se combina con un Router, que genera automáticamente las URLs por nosotros.

#### Paso 1: Simplificar las Vistas (api/views.py)

Sí, por supuesto. Esa es exactamente la forma más profesional y eficiente de hacerlo, y es el tema de esta lección.

Lo que hemos hecho hasta ahora ha sido el método “manual” para que entendieras las piezas. Ahora, vamos a usar la herramienta que Django REST Framework (DRF) provee para combinar todo el CRUD en una sola clase: el ViewSet.

## Lección 7: Refactorizando a ViewSets y Routers

Un ViewSet es una clase que agrupa toda la lógica de un recurso (en nuestro caso, las Tareas). Usaremos un ModelViewSet, que nos da toda la funcionalidad CRUD (list, create, retrieve, update, destroy) de forma gratuita.

Esto se combina con un Router, que genera automáticamente las URLs por nosotros.

Paso 1: Simplificar las Vistas (api/views.py) Vamos a reemplazar nuestras dos clases (TaskViewSet y TaskRetrieveUpdateDestroyAPIView) por una sola TaskViewSet.

* Abre api/views.py y reemplaza todo su contenido con esto: # api/views.py from rest\_framework import viewsets from .models import Task from .serializers import TaskSerializer
* class TaskViewSet(viewsets.ModelViewSet):  
   """  
   Una única ViewSet para ver, editar y eliminar tareas.  
   """  
   queryset = Task.objects.all()  
   serializer\_class = TaskSerializer

¡Y ya está! Esta única clase ahora maneja todas las operaciones.

#### Paso 2: Simplificar las URLs (api/urls.py)

Como un ViewSet no es una vista estándar, no podemos conectarlo con path(). Necesitamos un Router que genere las URLs por nosotros.

* Abre api/urls.py y reemplaza todo su contenido con esto: # api/urls.py from django.urls import path, include from rest\_framework.routers import DefaultRouter from .views import TaskViewSet
* # Crea un router y registra nuestro viewset con él.  
   router = DefaultRouter()  
   router.register(r'tasks', TaskViewSet, basename="task")  
    
   # Las URLs de la API son determinadas automáticamente por el router.  
   urlpatterns = [  
   path('', include(router.urls)),  
   ]

Este DefaultRouter genera automáticamente las mismas dos URLs que creamos antes a mano:

* /tasks/ (para listar y crear)
* /tasks// (para ver, actualizar y borrar)

¡Felicidades! Acabas de refactorizar tu código a la forma más común y recomendada de construir APIs con DRF. Has reemplazado dos vistas y dos rutas URL manuales por una ViewSet y un Router, logrando la misma funcionalidad con mucho menos código.

Si ahora pruebas tus endpoints (http://127.0.0.1:8000/api/tasks/ y http://127.0.0.1:8000/api/tasks/1/), verás que todo sigue funcionando exactamente igual.