

Webinar Python. El poder oculto detrás de las aplicaciones más exitosas

1. ¿Qué es Django?

Django es un **framework web en Python** que sigue el patrón **MTV (Model - Template - View: MTV es la versión de Django del patrón MVC)**, muy parecido al MVC tradicional. Está diseñado para el desarrollo rápido, seguro y escalable de aplicaciones web.

Flujo en Django (MTV)

```
Usuario → View (Controller) → Model → Template → Usuario
```

Flujo en MVC

```
Usuario → Controller → Model → View → Usuario
```

Cómo Django transforma MVC → MTV

Django sigue **la misma idea**, pero **renombra y reorganiza** las funciones:

MVC	Django MTV	Qué hace en Django
Model	Model	Igual que en MVC: define la estructura de datos y la lógica de negocio (en <code>models.py</code>)
View	Template	En lugar de "View", Django usa Template , que es la capa de presentación (HTML + etiquetas).
Controller	View	Django llama View a lo que en MVC sería el Controller : la función o clase que recibe la solicitud.

Ventajas principales:

- Arquitectura robusta y modular.
- ORM (Object-Relational Mapper) integrado.
- Panel de administración automático.
- Manejo nativo de seguridad (CSRF, XSS, inyección SQL).
- Escalable para proyectos grandes.
- Integración sencilla con **Bootstrap, APIs REST (Django REST Framework) y JavaScript/React/Vue**.
- Es un framework de backend (lado del servidor) en Python.

- Motor de Plantillas de Django (Django Template Language - DTL)
 - El frontend puede manejarse de tres formas principales:
 - Si buscas velocidad de desarrollo: Clásico/Monolítico: *DTL, Bootstrap o Tailwind CSS*
 - Interactividad: *DTL, + HTMX o Alpine.js*
 - Arquitectura desacoplada tipo API + SPA (Single Page Application): *React, Vue.js o Angular*
-

Comparativa Front con Django

Framework	Tipo de integración	Curva de aprendizaje	Ideal para...
Bootstrap	Plantillas HTML	Muy baja	Formularios, admin, sitios institucionales
Tailwind CSS	Plantillas HTML	Media	Diseño moderno, startups
React.js	API REST frontend	Alta	Aplicaciones SPA, SaaS, dashboards complejos
Vue.js	Integración parcial	Media	Apps interactivas moderadas
HTMX/Alpine	Plantillas HTML	Media	Apps Django con reactividad ligera

2. Instalación paso a paso

Prerrequisitos

Asegúrate de tener:

```
python --version
pip --version
```

*Si no los tienes, instala Python desde:

```
https://www.anaconda.com/download
```

Instalar Django

```
pip install django
```

Verificar versión:

```
django-admin --version
```

Instalar Postgres

Descargar desde:

<https://www.enterprisedb.com/downloads/postgres-postgresql-downloads>

3. Extensiones y herramientas útiles

Herramienta	Función	Enlace
<code>django-extensions</code>	Comandos extra (<code>shell_plus</code> , <code>show_urls</code> , etc.)	https://django-extensions.readthedocs.io
<code>django-crispy-forms</code>	Formularios con Bootstrap	https://django-crispy-forms.readthedocs.io
<code>django-debug-toolbar</code>	Depuración visual	https://django-debug-toolbar.readthedocs.io
<code>django-environ</code>	Manejo de variables de entorno	https://django-environ.readthedocs.io
<code>django-rest-framework</code>	Creación de APIs REST	https://www.django-rest-framework.org
<code>pytest-django</code>	Pruebas automáticas	https://pytest-django.readthedocs.io

Proyecto: Desarrollo completo paso a paso: Python +

- ▼ Django + PostgreSQL + APIs CRUD + Frontend con Bootstrap

Objetivo general: Los estudiantes construirán un proyecto funcional en Django que usa PostgreSQL como base de datos, expondrá APIs REST para operaciones CRUD y tendrá un frontend con Bootstrap que consuma esas APIs (operaciones crear, leer, actualizar, eliminar).

Requisitos previos (que deben confirmarse antes de iniciar)

- Python 3.11+ instalado y accesible desde la terminal (`python --version`).
- pip instalado (`pip --version`).
- PostgreSQL instalado localmente o acceso a una instancia (puede ser Docker).
Saber usuario, contraseña y nombre de BD.
- Conocimientos básicos de Python y HTML.
- Editor de código (VS Code recomendado) y terminal.

- ▼ 1. Instalar dependencias básicas

Librería	Función principal	Etapa donde se usa
<code>django</code>	Framework web base (MVC/MTV)	Todo el proyecto

Librería	Función principal	Etapa donde se usa
djangorestframework	Crear APIs REST	Desarrollo backend
psycopg2-binary	Conexión con PostgreSQL	Configuración de base de datos
gunicorn	Servidor WSGI (Web Server Gateway Interface) para producción	Despliegue
whitenoise	Servir archivos estáticos en producción	Despliegue

```
pip install django djangorestframework psycopg2-binary gunicorn whitenoise
```

2. Crear el proyecto Django y la app principal

```
django-admin startproject mi_proyecto #crear nuevo proyecto
cd mi_proyecto #cambiar de directorio
python manage.py startapp core #crear una nueva aplicación de Django
```

3. Abrir el proyecto en VS Code.

Estructura inicial generada a partir de la creación del proyecto con django-admin startproject mi_proyecto

```
proyecto/
|
|   manage.py
|
|   proyecto/
|       __init__.py
|       asgi.py
|       settings.py
|       urls.py
|       wsgi.py
```

- ◆ Nota: El primer `(proyecto/)` es la **carpeta raíz** del proyecto (que puedes renombrar). El segundo `(proyecto/)` (dentro) es el **paquete Python principal** con los archivos de configuración.

Carpetas y Archivos:

1. `manage.py`

Ubicación: `proyecto/manage.py`

Este archivo es el **punto de entrada principal** para interactuar con el proyecto Django desde la línea de comandos. Es como el “control remoto” de Django.

Funciones clave:

- Permite ejecutar comandos como:

```
python manage.py runserver  
python manage.py makemigrations  
python manage.py migrate  
python manage.py createsuperuser  
python manage.py startapp core
```

- Carga la configuración del proyecto (`settings.py`) para que Django sepa cómo comportarse.

👉 En resumen: `manage.py` **es el comando universal** para manejar cualquier tarea administrativa de Django.

2. `__init__.py`**Ubicación:** `proyecto/proyecto/__init__.py`

Es un archivo vacío que **marca la carpeta como un paquete Python**.

- Python necesita este archivo para tratar la carpeta `proyecto/` como un módulo importable.
- Si no estuviera, no podrías hacer cosas como `from proyecto import settings`.

💡 No se suele modificar. Su presencia es simplemente para estructurar correctamente el paquete.

3. `settings.py`**Ubicación:** `proyecto/proyecto/settings.py`

Es el **corazón del proyecto Django**. Aquí se definen todas las configuraciones principales del sistema.

▼ Contiene:

- Configuraciones generales (ruta base, clave secreta, modo debug, etc.)
- Aplicaciones instaladas (`INSTALLED_APPS`)

- Configuración de bases de datos (**DATABASES**)
- Configuración de idioma y zona horaria
- Configuración de archivos estáticos (**STATIC_URL**, etc.)

👉 En resumen: **define el comportamiento, los módulos y la infraestructura del proyecto.**

4. `urls.py`

Ubicación: `proyecto/proyecto/urls.py`

Contiene el **mapeo de URLs** hacia las vistas (views) del proyecto. Django utiliza este archivo para saber **qué función o clase ejecutar** cuando alguien accede a una URL específica.

Después podrás agregar tus propias rutas, por ejemplo:

👉 En resumen: `urls.py` es el mapa de rutas del sitio.

5. `wsgi.py`

Ubicación: `proyecto/proyecto/wsgi.py`

WSGI significa **Web Server Gateway Interface**. Este archivo define cómo el proyecto Django se comunicará con un **servidor web** (Apache, Nginx, Gunicorn, etc.) en producción.

👉 En resumen: `wsgi.py` se usa en despliegue (producción) para servir la aplicación Django en servidores WSGI compatibles.

6. `asgi.py`

Ubicación: `proyecto/proyecto/asgi.py`

ASGI significa **Asynchronous Server Gateway Interface**. Es el equivalente moderno de `wsgi.py`, pero permite manejar **conexiones asíncronas** como WebSockets o peticiones en tiempo real.

👉 En resumen: `asgi.py` se usa en servidores asíncronos (por ejemplo, Unicorn o Daphne) para manejar websocket o microservicios modernos.

▼ Resumen gráfico

Archivo	Función principal	Se usa en
<code>manage.py</code>	Ejecutar comandos del proyecto Django	Desarrollo y administración
<code>__init__.py</code>	Marca el paquete como módulo Python	Internamente
<code>settings.py</code>	Configuración global del proyecto	Siempre
<code>urls.py</code>	Mapa de rutas del sitio	Siempre
<code>wsgi.py</code>	Entrada para servidores web sincronizados	Producción
<code>asgi.py</code>	Entrada para servidores web asíncronos	Producción moderna (WebSockets)

Estructura inicial generada a partir de la creación del proyecto con python

manage.py startapp core

```
core/
|
├── __init__.py
├── admin.py
├── apps.py
├── migrations/
│   └── __init__.py
├── models.py
├── tests.py
└── views.py
```

Carpetas y Archivos:

1. `__init__.py`

Archivo vacío, marca la carpeta `core` como un **paquete Python**. Permite que Django y Python importen correctamente

 No se modifica normalmente.

2. `admin.py`

Archivo donde **registrarás tus modelos** para que aparezcan en el **panel de administración** de Django.

 En resumen: `admin.py` conecta tus tablas de base de datos (modelos) con la interfaz de administración (`/admin`).

3. `apps.py`

Define la **configuración de la aplicación** (nombre, etiquetas, comportamiento).

👉 En resumen: `apps.py` es la tarjeta de identidad de la aplicación dentro del ecosistema Django.

4. `migrations/`

Carpeta donde se guardan los **archivos de migración de base de datos**. Cada vez que creas o modificas un modelo y ejecutas:

💡 Esta carpeta permite a Django llevar **control histórico de los cambios en tus modelos**.

5. `models.py`

Aquí defines los **modelos**: las clases Python que representan las **tablas en la base de datos**.

👉 En resumen: `models.py` define la estructura de datos y las relaciones (entidades, campos, tipos).

6. `views.py`

Aquí se crean las **funciones o clases que procesan las peticiones HTTP** y devuelven respuestas (HTML, JSON, etc.).

👉 En resumen: `views.py` controla la lógica de negocio y lo que se devuelve al usuario o API.

7. `tests.py`

Archivo reservado para **pruebas automáticas**. Permite verificar que tus modelos, vistas y rutas funcionan correctamente.

👉 En resumen: `tests.py` garantiza que tu aplicación no se rompa cuando cambias código.

Resumen visual

Archivo	Función principal	Tipo de contenido
<code>__init__.py</code>	Marca la carpeta como paquete Python	Sistema

Archivo	Función principal	Tipo de contenido
admin.py	Registro de modelos para el panel de administración	Configuración
apps.py	Configura la aplicación en Django	Metadatos
migrations/	Control de versiones de base de datos	Estructural
models.py	Define las tablas de la BD (ORM)	Datos
views.py	Controla la lógica de negocio y respuestas	Lógica
tests.py	Automatiza pruebas del código	Calidad

4. Crear la base de datos en PostgreSQL (ejemplo con psql):

```
-- en la consola psql
CREATE DATABASE proyecto_db;
CREATE USER proyecto_user WITH PASSWORD 'TuPasswordSegura';
GRANT ALL PRIVILEGES ON DATABASE proyecto_db TO proyecto_user;
```

▼ 5. Configurar `proyecto/settings.py` (sección DATABASES)

```
"""
Django settings for mi_proyecto project.

Generated by 'django-admin startproject' using Django 5.2.7.

For more information on this file, see
https://docs.djangoproject.com/en/5.2/topics/settings/

For the full list of settings and their values, see
https://docs.djangoproject.com/en/5.2/ref/settings/
"""

from pathlib import Path

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent

# Quick-start development settings - unsuitable for production
# See https://docs.djangoproject.com/en/5.2/howto/deployment/checklist/

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = "django-insecure-gb^w%x0+p8eh!fn!iq9$hog9pdj*n%qz^#sx7j7p@&d^_z&m"

# SECURITY WARNING: don't run with debug turned on in production!
```

```
DEBUG = True

ALLOWED_HOSTS = []

# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    'rest_framework',
    'core',
]

MIDDLEWARE = [
    "django.middleware.security.SecurityMiddleware",
    "django.contrib.sessions.middleware.SessionMiddleware",
    "django.middleware.common.CommonMiddleware",
    "django.middleware.csrf.CsrfViewMiddleware",
    "django.contrib.auth.middleware.AuthenticationMiddleware",
    "django.contrib.messages.middleware.MessageMiddleware",
    "django.middleware.clickjacking.XFrameOptionsMiddleware",
]

ROOT_URLCONF = "mi_proyecto.urls"

TEMPLATES = [
{
    "BACKEND": "django.template.backends.django.DjangoTemplates",
    'DIRS': [BASE_DIR / 'templates'],
    "APP_DIRS": True,
    "OPTIONS": {
        "context_processors": [
            'django.template.context_processors.debug',
            'django.template.context_processors.request',
            'django.contrib.auth.context_processors.auth',
            'django.contrib.messages.context_processors.messages',
        ],
    },
},
],
]

WSGI_APPLICATION = "mi_proyecto.wsgi.application"

# Database
# https://docs.djangoproject.com/en/5.2/ref/settings/#databases
```

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': ' proyecto_db',
        'USER': 'postgres',
        'PASSWORD': 'postgres',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}

# Password validation
# https://docs.djangoproject.com/en/5.2/ref/settings/#auth-password-validators

AUTH_PASSWORD_VALIDATORS = [
    {
        "NAME": "django.contrib.auth.password_validation.UserAttributeSimilarityValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.MinimumLengthValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.CommonPasswordValidator",
    },
    {
        "NAME": "django.contrib.auth.password_validation.NumericPasswordValidator",
    },
]

# Internationalization
# https://docs.djangoproject.com/en/5.2/topics/i18n/

LANGUAGE_CODE = "en-us"

TIME_ZONE = "UTC"

USE_I18N = True

USE_TZ = True

# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/5.2/howto/static-files/

STATIC_URL = "static/"
STATICFILES_DIRS = [BASE_DIR / 'static']

# Default primary key field type
```

```
# https://docs.djangoproject.com/en/5.2/ref/settings/#default-auto-field
```

```
DEFAULT_AUTO_FIELD = "django.db.models.BigAutoField"
```

▼ 6. Modelos, migraciones y admin

Vamos a crear un ejemplo clásico: *Gestión de tareas* (Task).

1. core/models.py

```
# Importamos el módulo 'models' de Django, que contiene todas las clases base
# necesarias para definir modelos (tablas en la base de datos).
from django.db import models

# Definimos la clase Task, que hereda de models.Model.
# Cada clase de este tipo representa una tabla en la base de datos.
class Task(models.Model):

    # Campo de texto corto (máximo 200 caracteres).
    # Se usa para el título de la tarea.
    title = models.CharField(max_length=200)

    # Campo de texto largo, ideal para descripciones o detalles adicionales.
    # 'blank=True' permite que este campo sea opcional en formularios.
    description = models.TextField(blank=True)

    # Campo booleano para marcar si la tarea está completada o no.
    # Por defecto, toda tarea nueva se crea como no completada.
    completed = models.BooleanField(default=False)

    # Fecha y hora de creación. Se asigna automáticamente al crear el registro.
    created_at = models.DateTimeField(auto_now_add=True)

    # Fecha y hora de la última actualización. Se actualiza cada vez que se guarda.
    updated_at = models.DateTimeField(auto_now=True)

    # Este método define cómo se mostrará el objeto cuando se imprima o liste en la consola.
    def __str__(self):
        return self.title
```

7. Migraciones iniciales y comprobar conexión

Ejecutar los siguientes comandos en la siguiente ruta:

```
\mi_proyecto\
```

- `python manage.py makemigrations:`

Este comando crea los archivos de migración (carpeta Migrations) basados en Traduce los cambios en tu código Python (los modelos) a instrucciones que D

```
python manage.py makemigrations
```

- `python manage.py migrate:`

Este comando aplica los cambios (migraciones) a la base de datos real. Ejecuta las instrucciones generadas por makemigrations para crear, modifica

```
python manage.py migrate
```

Análisis de las tablas creadas en Postgres

Tablas de Django por defecto:

- `auth_group` - Grupos de usuarios
- `auth_group_permissions` - Permisos de grupos
- `auth_permission` - Permisos del sistema
- `auth_user` - Usuarios del sistema
- `auth_user_groups` - Relación usuarios-grupos
- `auth_user_user_permissions` - Permisos de usuarios
- `django_admin_log` - Log del panel de administración
- `django_content_type` - Tipos de contenido
- `django_migrations` - Registro de migraciones aplicadas
- `django_session` - Sesiones de usuarios

Tu tabla personalizada:

- `core_task` - ¡Esta es tu tabla! Creada por tu modelo `Task`

Si no hay errores, la conexión con Postgres funciona.

8. Crear superusuario y comprobar admin

```
python manage.py createsuperuser
```

¿Es necesario?

- Sí, si quieres acceder al panel de administración de Django
- NO, si solo vas a usar la API o la aplicación sin admin

¿Para qué sirve?

- Crea un usuario administrador para acceder a <http://127.0.0.1:8000/admin>
- Te permite gestionar tus modelos desde la interfaz web de Django
- Es muy útil para ver, crear, editar y eliminar registros de la base de datos

```
python manage.py createsuperuser # ingresar usuario/email/password
```

```
python manage.py runserver
```

¿Es necesario?

- Sí, si quieres probar tu aplicación
- NO, si solo estás configurando la base de datos

¿Para qué sirve?

- Inicia el servidor de desarrollo de Django
- Te permite acceder a tu aplicación en <http://127.0.0.1:8000>
- Necesario para probar tanto el admin como cualquier API que crees

Ruta: Dentro del proyecto en la carpeta donde se encuentre `manage.py`

```
python manage.py runserver # abrir http://127.0.0.1:8000/admin
```

Registrar la app en `settings.py` para crear migraciones y aplicarlas

- `manage.py makemigrations core`: Verificará si hay cambios en la app `core`

```
python manage.py makemigrations core
```

- Se ejecuta de nuevo

```
python manage.py migrate
```

✓ Registrar el modelo en el admin `core/admin.py`

Registrar un modelo significa decirle a Django que ese modelo debe aparecer y poder administrarse en el **Panel de administración web (/admin)**. Sin este registro, aunque el modelo exista en la base de datos, no aparecerá en la **interfaz del administrador (Panel) de Django**.

```
# Importa el módulo 'admin' que permite registrar y personalizar la interfaz
# de administración de Django (el panel que se accede en /admin).
from django.contrib import admin

# Importa el modelo 'Task' desde el archivo models.py del mismo módulo (app act
from .models import Task

# Usa el decorador '@admin.register' para registrar el modelo 'Task'
# directamente en el panel de administración.
# Esto evita tener que escribir admin.site.register(Task, TaskAdmin)
@admin.register(Task)
class TaskAdmin(admin.ModelAdmin):
    """
    Esta clase personaliza la forma en que el modelo Task se muestra
    y se gestiona dentro del panel de administración de Django.
    Hereda de admin.ModelAdmin, que ofrece opciones avanzadas de configuración.
    """

    # Define qué campos se mostrarán en la lista del panel de administración.
    # Es decir, las columnas visibles en la tabla de tareas.
    list_display = ('id', 'title', 'completed', 'created_at')

    # Permite agregar un filtro lateral por el campo 'completed' (True/False)
    # para que el administrador pueda filtrar fácilmente las tareas completadas
    list_filter = ('completed',)

    # Habilita una barra de búsqueda en la parte superior del panel de tareas,
    # permitiendo buscar por 'title' o 'description'.
    search_fields = ('title', 'description')
```

Verificar en el admin que se puede crear tareas.

✓ 9. APIs REST con Django REST Framework

1. Crear `core/serializers.py`

Serializer

- Serializar = transformar un objeto complejo en un formato simple que pueda viajar por la red o guardarse fácilmente
- Se encarga de convertir objetos de Django (modelos) en JSON, y viceversa.

```
# Importamos el módulo serializers del framework DRF.
# Este módulo contiene clases que permiten convertir (serializar y deserializar
# objetos de Django a formatos como JSON o XML.
from rest_framework import serializers

# Importamos el modelo Task que definimos previamente en models.py.
# Este modelo será la base para construir el serializer.
from .models import Task

# Definimos una clase que hereda de serializers.ModelSerializer,
# una clase del DRF que simplifica la creación de serializers basados en modelos.
class TaskSerializer(serializers.ModelSerializer):

    # La clase interna Meta define cómo se comportará el serializer.
    class Meta:
        # Especificamos cuál es el modelo que queremos convertir a JSON.
        model = Task

        # Con 'fields = "__all__"' indicamos que queremos incluir TODOS los campos
        # del modelo (id, title, description, completed, created_at, etc.).
        # Si quisieramos solo algunos, podríamos usar: fields = ['title', 'completed']
        fields = '__all__'
```

2. Crear vistas (usaremos viewsets) (Controlador) `core/views.py`

```
# Importamos las clases y módulos necesarios de Django REST Framework (DRF)
from rest_framework import viewsets # viewsets permite crear vistas CRUD completamente

# Importamos el modelo que queremos exponer mediante la API
from .models import Task

# Importamos el serializer que convierte los objetos Task a JSON y viceversa
from .serializers import TaskSerializer

# Definimos una clase que hereda de viewsets.ModelViewSet
# ModelViewSet es una clase especial de DRF que automáticamente
# crea todas las operaciones CRUD:
# - GET (listar o ver una tarea)
# - POST (crear nueva tarea)
```

```
# - PUT / PATCH (actualizar tarea existente)
# - DELETE (eliminar tarea)
class TaskViewSet(viewsets.ModelViewSet):

    # 'queryset' define qué registros del modelo se van a manejar.
    # Aquí obtenemos todas las tareas y las ordenamos de más reciente a más antigua
    queryset = Task.objects.all().order_by('-created_at')

    # 'serializer_class' indica qué serializer se usará para transformar los datos
    # (de modelo a JSON y de JSON a modelo)
    serializer_class = TaskSerializer
```

3. Ruteo de la API `core/urls.py`

- Define las rutas (endpoints) específicas de la aplicación core, es decir, los caminos por los que los usuarios o sistemas externos acceden a tu API

```
# Importamos las funciones necesarias del módulo de URLs de Django
from django.urls import path, include

# Importamos el sistema de enrutamiento de Django REST Framework (DRF)
# 'routers' nos permite registrar ViewSets de manera automática sin definir cada una
from rest_framework import routers

# Importamos el ViewSet que creamos para el modelo Task
from .views import TaskViewSet

# Creamos una instancia del router por defecto de DRF.
# Este router se encargará de generar automáticamente las rutas CRUD
# (GET, POST, PUT, DELETE) para el TaskViewSet.
router = routers.DefaultRouter()

# Registraremos el ViewSet 'TaskViewSet' dentro del router.
# - El primer argumento ('r"tasks"') define la ruta base de la API: /api/tasks/
# - El segundo argumento es la vista (TaskViewSet)
# - 'basename' se usa internamente por DRF para generar nombres únicos de rutas
router.register(r'tasks', TaskViewSet, basename='task')

# Definimos la lista de patrones de URL (urlpatterns)
urlpatterns = [
    # Incluimos todas las rutas generadas automáticamente por el router
    # bajo el prefijo 'api/'. Esto significa que las URLs finales serán:
    # /api/tasks/      → lista y creación
    # /api/tasks/<id>/ → detalle, actualización o eliminación
    path('api/', include(router.urls)),
```

]

Y enlazar `proyecto/urls.py`

- Permite enlazar las rutas de la API con el archivo principal del proyecto

```
# Importamos las funciones necesarias del sistema de enrutamiento de Django.  
# 'path' se usa para definir rutas y 'include' permite incluir las rutas de otra aplicación  
from django.contrib import admin  
from django.urls import path, include  
  
# Definimos la lista principal de rutas (urlpatterns)  
urlpatterns = [  
    # Ruta para el panel de administración de Django  
    # Accedes a él desde: http://127.0.0.1:8000/admin/  
    path('admin/', admin.site.urls),  
  
    # Incluimos todas las rutas definidas en la aplicación 'core'  
    # El prefijo '' significa que las rutas de core estarán disponibles directamente  
    # Por ejemplo:  
    # /api/tasks/ → rutas definidas en core/urls.py  
    path('', include('core.urls')),  
]
```

4. Probar la API con `runserver` y acceder a `http://127.0.0.1:8000/api/tasks/` (list/create).

Endpoints disponibles:

- `GET http://127.0.0.1:8000/api/tasks/` - Listar todas las tareas
- `POST http://127.0.0.1:8000/api/tasks/` - Crear nueva tarea
- `GET http://127.0.0.1:8000/api/tasks/{id}/` - Obtener tarea específica
- `PUT/PATCH http://127.0.0.1:8000/api/tasks/{id}/` - Actualizar tarea
- `DELETE http://127.0.0.1:8000/api/tasks/{id}/` - Eliminar tarea

▼ 10. Frontend con Bootstrap que consume las APIs (CRUD)

Ahora, se crean las plantillas sencillas y se usará `fetch` desde JavaScript para la comunicación con la API.

Estructura de archivos a crear:

1. Crear la carpeta de plantillas:

```
mi_proyecto/
└── templates/
    └── core/
        ├── base.html
        └── index.html
```

2. `templates/core/base.html` (head mínimo con Bootstrap CDN)

- Necesitamos crear la estructura de una página web, para ello, escrimos "!" + la tecla `tab`, eso nos devuelve:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Prueba</title>
</head>
<body>
    <h1>Prueba</h1>
</body>
</html>
```

- Luego, visitamos la web de Bootstrap para integrar el "link" y el "script" de Bootstrap. El código completo queda de la siguiente forma:

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Prueba</title>
    <link
        href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css"
        rel="stylesheet"
        integrity="sha384-sRI14kxILFvY47J16cr9ZwB07vP4J8+LH7qKQnuqkuIAvNWLzeN8tE5
        crossorigin="anonymous"
    />
</head>
<body>
    <h1>Prueba</h1>
```

```
<script>
  src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.bundle.min.js"
  integrity="sha384-FKyoEForCGlyvwx9Hj09JcYn3nv7wiPVlz7YYwJrWVcXK/BmnVDxM+D
  crossorigin="anonymous"
</script>
</body>
</html>
```

3. Seguidamente, ubicamos una Barra de navegación de Bootstrap y la integramos en "base.html". El código completo queda de la siguiente forma:

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title></title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css" rel="stylesheet"
      integrity="sha384-sRI14kxILFvY47J16cr9ZwB07vP4J8+LH7qKQnuqkuIAvNWLzeN8tE5
      crossorigin="anonymous" />
  </head>
  <body>
    <nav class="navbar bg-body-tertiary">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">To Do App</a>
      </div>
    </nav>

    <script>
      src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.bundle.min.js"
      integrity="sha384-FKyoEForCGlyvwx9Hj09JcYn3nv7wiPVlz7YYwJrWVcXK/BmnVDxM+D
      crossorigin="anonymous"
    </script>

  </body>
</html>
```

4. Seguidamente, integramos:

```
<div class="container">{% block content %}{% endblock %}</div>
```

Que es una clase de Bootstrap que crea un contenedor centrado y con márgenes automáticos que contiene una etiqueta de bloque de Django, usada para insertar contenido dinámico dentro de una plantilla base.

y

```
{% block extra_js %}{% endblock %}
```

Permite a cada página incluir su propio código JS, sin ensuciar ni duplicar la plantilla principal

```
<!DOCTYPE html>
<html lang="es">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title></title>
    <link
      href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/css/bootstrap.min.css"
      rel="stylesheet"
      integrity="sha384-sRIl4kxILFvY47J16cr9ZwB07vP4J8+LH7qKQnuqkuIAvNWLzeN8tE5"
      crossorigin="anonymous"
    />
  </head>
  <body>
    <nav class="navbar navbar-expand-lg bg-body-tertiary">
      <div class="container-fluid">
        <a class="navbar-brand" href="#">To Do App</a>
      </div>
    </nav>

    <div class="container">{% block content %}{% endblock %}</div>

    <script
      src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.8/dist/js/bootstrap.bundle.min.js"
      integrity="sha384-FKyoEForCGlyvwx9Hj09JcYn3nv7wiPVlz7YYwJrWVcXK/BmnVDxM+C"
      crossorigin="anonymous"
    ></script>

    {% block extra_js %}{% endblock %}
  </body>
</html>
```

3. `templates/core/index.html` (lista + formulario modal para crear/editar)

```
{% extends 'core/base.html' %} {% block content %}  
<h1>Lista de Tareas</h1>  
<div id="alert-placeholder"></div>  
<button class="btn btn-primary mb-3" id="btn-new">Nueva tarea</button>  
<table class="table table-striped">  
    <thead>  
        <tr>  
            <th>ID</th>  
            <th>Título</th>  
            <th>Completada</th>  
            <th>Acciones</th>  
        </tr>  
    </thead>  
    <tbody id="tasks-body"></tbody>  
</table>  
  
<div class="modal fade" id="taskModal" tabindex="-1">  
    <div class="modal-dialog">  
        <div class="modal-content">  
            <div class="modal-header">  
                <h5 class="modal-title" id="modalTitle">Nueva tarea</h5>  
                <button  
                    type="button"  
                    class="btn-close"  
                    data-bs-dismiss="modal"  
                ></button>  
            </div>  
            <div class="modal-body">  
                <form id="taskForm">  
                    <input type="hidden" id="taskId" />  
                    <div class="mb-3">  
                        <label for="title">Título</label>  
                        <input id="title" class="form-control" required />  
                    </div>  
                    <div class="mb-3">  
                        <label for="description">Descripción</label>  
                        <textarea id="description" class="form-control"></textarea>  
                    </div>  
                    <div class="form-check">  
                        <input type="checkbox" id="completed" class="form-check-input" />  
                        <label class="form-check-label" for="completed">Completada</label>  
                    </div>  
                </form>  
            </div>  
            <div class="modal-footer">  
                <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">  
                    Cerrar  
                </button>  
                <button type="button" class="btn btn-primary" id="saveBtn">  
                    Guardar  
                </button>  
            </div>  
        </div>  
    </div>  
</div>
```

```
</div>
</div>
</div>
</div>
{%- endblock %}

{% block extra_js %}
<script>
    // 🔵 URL base de la API REST (definida en Django REST Framework)
    // Aquí apuntamos al endpoint /api/tasks/ para consumir los datos
    const apiBase = "/api/tasks/";

    // =====
    // ✅ Función para obtener todas las tareas (READ)
    // =====
    async function fetchTasks() {
        // Realiza una petición GET a la API
        const res = await fetch(apiBase);
        // Convierte la respuesta a formato JSON
        const data = await res.json();

        // Selecciona el cuerpo de la tabla donde se mostrarán las tareas
        const body = document.getElementById("tasks-body");
        // Limpia el contenido previo antes de renderizar nuevamente
        body.innerHTML = "";

        // Recorre todas las tareas recibidas desde el backend
        data.forEach((t) => {
            // Crea una fila <tr> para la tabla
            const tr = document.createElement("tr");
            // Inserta el contenido HTML de la fila con los datos de la tarea
            tr.innerHTML = `
                <td>${t.id}</td>
                <td>${t.title}</td>
                <td>${t.completed ? "Sí" : "No"}</td>
                <td>
                    <button class="btn btn-sm btn-info" onclick="editTask(${t.id})">Editar</button>
                    <button class="btn btn-sm btn-danger" onclick="deleteTask(${t.id})">Eliminar</button>
                </td>`;
            // Agrega la fila al cuerpo de la tabla
            body.appendChild(tr);
        });
    }

    // =====
    // ✅ Crear nueva tarea (CREATE)
    // =====
```

```
async function createTask(payload) {
    await fetch(apiBase, {
        method: "POST", // Método HTTP POST para crear
        headers: { "Content-Type": "application/json" }, // Indicamos formato JSON
        body: JSON.stringify(payload), // Convertimos el objeto a JSON
    });
}

// =====
// ✅ Actualizar una tarea existente (UPDATE)
// =====
async function updateTask(id, payload) {
    await fetch(` ${apiBase}${id}/`, {
        method: "PUT", // Método HTTP PUT para actualizar
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(payload),
    });
}

// =====
// ✅ Eliminar tarea (DELETE)
// =====
async function deleteTask(id) {
    // Confirma antes de eliminar
    if (!confirm("¿Eliminar tarea?")) return;
    await fetch(` ${apiBase}${id}/`, { method: "DELETE" });
    // Vuelve a cargar la lista de tareas
    fetchTasks();
}

// =====
// ✅ Mostrar alertas temporales (Bootstrap)
// =====
function showAlert(message, type = "success") {
    const p = document.getElementById("alert-placeholder");
    // Inserta un mensaje de alerta dinámico en el contenedor
    p.innerHTML = `<div class="alert alert-${type}" role="alert">${message}</div>`;
    // Borra el mensaje después de 3 segundos
    setTimeout(() => (p.innerHTML = ""), 3000);
}

// =====
// ✅ Manejo del modal (Bootstrap)
// =====
const modal = new bootstrap.Modal(document.getElementById("taskModal"));

// Cuando se hace clic en el botón “Nueva tarea”
document.getElementById("btn-new").addEventListener("click", () => {
    document.getElementById("modalTitle").innerText = "Nueva tarea"; // Cambia
    document.getElementById("taskId").value = ""; // Limpia ID
    document.getElementById("taskForm").reset(); // Limpia formulario
})
```

```
modal.show(); // Muestra el modal
});

// =====
// ✅ Cargar datos en el modal para editar tarea existente
// =====
async function editTask(id) {
  const res = await fetch(`${apiBase}${id}/`); // Pide datos de la tarea
  const t = await res.json(); // Convierte la respuesta en JSON

  // Rellena los campos del formulario con los datos existentes
  document.getElementById("taskId").value = t.id;
  document.getElementById("title").value = t.title;
  document.getElementById("description").value = t.description;
  document.getElementById("completed").checked = t.completed;

  document.getElementById("modalTitle").innerText = "Editar tarea"; // Cambia
  modal.show(); // Muestra el modal
}

// =====
// ✅ Guardar tarea (crear o actualizar)
// =====
document.getElementById("saveBtn").addEventListener("click", async () => {
  // Obtiene el ID (vacío si es una tarea nueva)
  const id = document.getElementById("taskId").value;

  // Construye el objeto de datos a enviar
  const payload = {
    title: document.getElementById("title").value,
    description: document.getElementById("description").value,
    completed: document.getElementById("completed").checked,
  };

  try {
    // Si hay ID → actualizar, si no → crear nueva
    if (id) await updateTask(id, payload);
    else await createTask(payload);

    modal.hide(); // Cierra el modal
    showAlert("Guardado correctamente"); // Muestra mensaje de éxito
    fetchTasks(); // Recarga la lista
  } catch (err) {
    showAlert("Error al guardar", "danger"); // Muestra mensaje de error
  }
});

// =====
// ✅ Inicialización
// Llama a fetchTasks() apenas se carga la página
// =====
```

```
fetchTasks();
</script>
{% endblock %}
```

4. Crear vista que renderiza `index.html` en `core/views.py` (añadir import):

```
from django.shortcuts import render

def index(request):
    return render(request, 'core/index.html')
```

5. Añadir ruta en `core/urls.py` antes del `include(router.urls)`:

```
from django.urls import path, include
from rest_framework import routers
from .views import TaskViewSet, index

router = routers.DefaultRouter()
router.register(r'tasks', TaskViewSet, basename='task')

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

5. Comprobar funcionamiento: abrir `http://127.0.0.1:8000/` y probar crear, editar, eliminar.

Buenas prácticas y consideraciones (rápidas)

- Validar datos en el serializer (campos `required`, `max_length`).
- Manejar permisos y autenticación (Token / Session) para APIs en producción.
- Usar paginación en listas grandes: `REST_FRAMEWORK = {
 'DEFAULT_PAGINATION_CLASS':
 'rest_framework.pagination.PageNumberPagination', 'PAGE_SIZE': 10 }
 }` en `settings.py`.
- Evitar exponer `psycopg2-binary` en producción (usar `psycopg2` y un proceso de compilación), y mover credenciales a variables de entorno.

