

## Primer proyecto con Django Rest Framework

1. Antes que cualquier otra cosa, **creamos una carpeta** donde se guardará el proyecto, la que abriremos en VSC (Visual Studio Code).

2. **Creamos** nuestro entorno virtual

**TERMINAL ->** `python -m venv nombre_ambiente`

3. **Activamos** el entorno Virtual

**TERMINAL ->** `.\nombre_ambiente\Scripts\activate`

*En esta parte PUEDE que aparezca un error de activación del Entorno Virtual, pero se soluciona con un par de líneas en el terminal.*

*SI NO TIENE UN ERROR HASTA AQUI, NO ES NECESARIO EJECUTAR ESTAS LINEAS Y PUEDE CONTINUAR CON LA SIGUIENTE SECCION.*

**TERMINAL ->** `Set-ExecutionPolicy Unrestricted -Force`

**TERMINAL ->** `Set-ExecutionPolicy Unrestricted -Scope Process`

Podemos encontrar más información sobre esta solución, en la página de Microsoft: [ESTABLECER POLÍTICAS DE EJECUCIÓN](#)

4. Actualizamos PIP en el terminal

**TERMINAL ->** `python.exe -m pip install --upgrade pip`

5. Hacemos una instalación de Django

**TERMINAL ->** `pip install Django`

6. Creamos el proyecto Django Rest Framework.

**TERMINAL ->** `django-admin startproject drf .`

Es importante el punto al final de la instrucción, para que nos cree la carpeta/proyecto en el mismo nivel que *nombre\_ambiente*. Aquí se crean los archivos básicos para el proyecto Django.

7. Creamos la aplicación *nombre\_aplicacion*, que incluirá los modelos (models.py para BD), vistas (views.py para las páginas) y el administrador (admin.py).

**TERMINAL** -> *django-admin startapp nombre\_aplicacion*

8. Dentro del directorio de Django, buscamos el archivo **settings.py** y lo editamos. Agregamos una línea para indicar que hemos creado una nueva aplicación llamada *nombre\_aplicacion* en la sección *INSTALLED\_APPS*. Recuerde respetar la separación de los elementos con una coma, ya que lo que tenemos es una lista.

```
# Application definition

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

9. A continuación, creamos nuestro modelo (BD) en el archivo **models.py**  
Con esto agregamos las tablas, los campos y tipos de datos al modelo.

```

# Principales tipos de datos para definir modelos en Gjango.
# En cada campo se puede indicar si admite valores nulos
# En cada campo se puede indicar valores por defecto

class Modelo_1(models.Model):
    # Campo de tipo AUTO INCREMENTAL, usado para ID's
    campo_1_modelo_1 = models.AutoField()
    # CharField para textos cortos y medianos, requiere indicar el largo máximo del campo
    campo_2_modelo_1 = models.CharField(max_length=200, null=False)
    # TextField para textos largos
    campo_3_modelo_1 = models.TextField(default='Texto')

class Modelo_2(models.Model):
    # Campo de tipo AUTO INCREMENTAL
    campo_1_modelo_2 = models.AutoField()
    # Campo de tipo RELACIÓN, cada objeto de Modelo_2 está relacionado con un objeto de Modelo_1
    campo_2_modelo_2 = models.ForeignKey(Modelo_1, on_delete=models.CASCADE)
    # Campo tipo FECHA
    campo_3_modelo_2 = models.DateField()
    # Campo tipo HORA
    campo_4_modelo_2 = models.TimeField()
    # Campo tipo FECHA con HORA
    campo_5_modelo_2 = models.DateTimeField()
    # Campo tipo ENTERO, puede variar entre SmallIntegerField o BigIntegerField
    campo_6_modelo_2 = models.IntegerField()
    # Campo tipo DECIMAL, se puede indicar cantidad máxima de dígitos y cantidad de decimales
    campo_7_modelo_2 = models.DecimalField()
    # Campo tipo DECIMAL
    campo_8_modelo_2 = models.FloatField()
    # Campo tipo BOOLEANO
    campo_9_modelo_2 = models.BooleanField()
    # Campo tipo CORREO
    campo_10_modelo_2 = models.EmailField()
    # Campo tipo URL
    campo_11_modelo_2 = models.URLField()
    # Campo tipo ARCHIVO
    campo_12_modelo_2 = models.URLField()
    # Campo tipo IMAGEN
    campo_13_modelo_2 = models.ImageField()

```

10. Luego importamos los modelos al archivo **admin.py** para dejar que podamos ver los modelos desde nuestro panel de administración.

**VSC** -> from .models import Modelo\_1

11. También registramos los modelos dentro del archivo **admin.py**, después de la línea "# Register your models here."

**VSC** -> admin.site.register(Modelo\_1)

12. Esto es OPCIONAL, ejecutamos la lista de programas instalados en el ENV para verificar que todo este OK y tengamos los programas necesarios instalados.

**TERMINAL ->** pip list

13. Ejecutamos las migraciones del modelo a la BD, por lo que debemos instalar el conector requerido, para MySQL en este caso.

- a. TERMINAL -> pip install mysqlclient
- b. Usando nuestra forma preferida, creamos la DB.
- c. Una vez con la base de datos creada, procedemos a modificar nuestro archivo settings.py. Indicamos que haremos uso del gestor MySQL.

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

```
DATABASES = {  
    'default': {  
        'ENGINE': 'django.db.backends.mysql',  
        'NAME': 'nombre_mi_db',  
        'USER': 'root',  
        'PASSWORD': '',  
        'HOST': 'localhost',  
        'PORT': '3306',  
    }  
}
```

14. Ejecutamos todos los cambios especificados en la DB.

**TERMINAL ->** python manage.py migrate

15. Se crean las migraciones para los modelos definidos.

**TERMINAL ->** python manage.py makemigrations

16. Se aplican la migraciones dentro de la BD.

**TERMINAL ->** python manage.py migrate

17. Si es necesario agregar algún datos a la DB una vez aplicadas las migraciones, Django considera el siguiente método que asegura contar con los mismos datos iniciales en cualquier ambiente de trabajo.

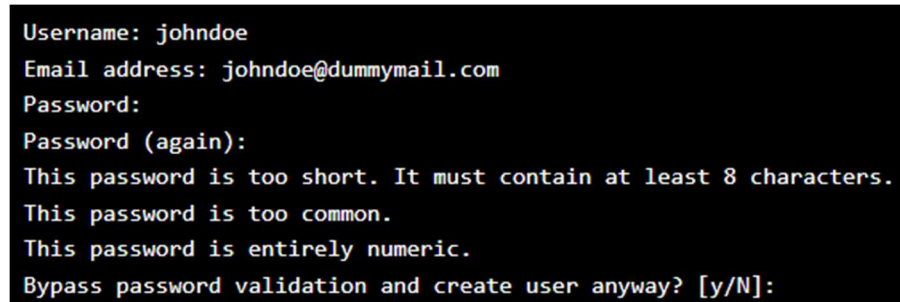
Primero debemos crear una carpeta fixtures dentro de nuestra aplicación. En esta carpeta crearemos un archivo donde pondremos los datos formateados en alguno de estas opciones: JSON, XML o YAML, con su correspondiente extensión, luego se ejecutará el siguiente comando para insertar esa data en la DB:

**TERMINAL ->** `python manage.py loaddata mi_archivo_con_datos`

18. Creamos el super usuario en el terminal, con esto ya podremos luego ingresar al administrador que estamos creando.

Usted debe ingresar los datos pedidos en el prompt del terminal, Si utiliza una contraseña insegura, verá una ADVERTENCIA y deberá usar una más fuerte en ambiente de producción, aunque se puede obviar en desarrollo.

**TERMINAL ->**`python manage.py createsuperuser`



```
Username: johndoe
Email address: johndoe@dummymail.com
Password:
Password (again):
This password is too short. It must contain at least 8 characters.
This password is too common.
This password is entirely numeric.
Bypass password validation and create user anyway? [y/N]:
```

Acá debemos confirmar con 'Y' la creación del usuario.

19. Iniciamos el server con el siguiente comando:

**TERMINAL ->** `python manage.py runserver`

Con esto ya tenemos nuestro SERVER corriendo Django con su proyecto y su BD.

Ahora podemos ir a nuestro navegador y escribir la ruta que nos da el terminal (<http://127.0.0.1:8000/>) para probar que estemos visibles y tenemos acceso al modelo que creamos.

Luego, detenemos el servidor desde el terminal con las teclas CONTROL+C para continuar con DJANGO REST FRAMEWORK.

1. Ahora instalamos DjangoRestFramework en nuestro terminal

**TERMINAL** -> pip install djangorestframework

2. Actualizamos el archivo **settings.py** y agregamos una línea en las INSTALLED\_APPS.

**VSC** -> 'rest\_framework',

3. Ahora vamos a proveer los datos a la APLICACIÓN, para eso vamos a serializar los datos en formato JSON.

Creamos en la carpeta de la aplicación un archivo llamado **serializer.py** (según la documentación de DJANGO en su página web).

**VSC** -> from rest\_framework import serializers

from .models import programmer

class Modelo\_1\_Serializer(serializers.ModelSerializer):

class Meta:

model = Modelo\_1

# fields = ('fullname','lenguaje','is\_active') acá podemos traer cualquier atributo o campo del modelo.

fields = '\_\_all\_\_'

# con la opción '\_\_all\_\_' traemos todos los campos para ver y tener acceso a todo el registro de cada elemento que pertenezca al modelo

Con esta clase ya tenemos las vistas del Modelo necesarias para realizar los métodos CRUD.

4. Ahora vamos a modificar el archivo **views.py** de la carpeta API.

Ingresamos el siguiente código.

**VSC ->** # from django.shortcuts import render <-- esta libreria no la usaremos por ahora

```
from rest_framework import viewsets
from .serializer import ProgrammerSerializer
from .models import programmer
# Create your views here.
class Modelo_1_ViewSet(viewsets.ModelViewSet):
    # acá creamos una QUERY a nuestra tabla, trayendo
    todos los campos como un objeto.
    queryset = programmer.objects.all()
    # Agregamos la clase ProgrammerSerializer que ya
    tiene el modelo serializado para mostrar
    serializer_class = Modelo_1_Serializer
```

5. Creamos otro archivo dentro de la carpeta de la aplicación con el nombre **urls.py** y ponemos este código.

**VSC ->** from django.urls import path, include

```
from rest_framework import routers
from api import views

router = routers.DefaultRouter() # este elemento
enrutador permite manejar múltiples rutas.

# esta es la base del conjunto de rutas o la raíz de las
rutas

# acá se manejan las rutas o ENDsPOINTS que pueda
tener tu API

router.register(r'programmers',
views.ProgrammerViewSet)

# la r permite que no se interprete como un salto de
línea o como un escape de carácter
```

```
# usamos la r para indicar que no tome los caracteres
como \n o \t que es un salto de línea o una tabulación,
es un formato tipo RAW de python.

# 'programmers' es un ENDPOINT

urlpatterns = [

    path("", include(router.urls))

# la ruta base va a incluir todos los elementos que tenga
el router que hemos creado en URLS

# esta es la lista de URLS que maneja ROUTER en sus
elementos URLS

]
```

6. Ahora incluimos o agregamos estas URLS en las vistas del proyecto (**url.py** de la carpeta drf)

**VSC ->** from django.contrib import admin

```
from django.urls import path, include

urlpatterns = [

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

    path('nombre_aplicacion/', include('
nombre_aplicacion.urls'))

]
```

Con esto seria todo en cuanto a configuración de nuestra aplicación en Django Rest Framework

Ahora podemos probar todo en las siguientes rutas:

<http://127.0.0.1:8000/>

[http://127.0.0.1:8000/nombre\\_aplicacion/](http://127.0.0.1:8000/nombre_aplicacion/)

<http://127.0.0.1:8000/admin/>



## Definición Página de Inicio Aplicación

Al haber hecho estas serializaciones y haber creado nuestras vistas de clases CBV (Class Based View) perdemos el acceso a la página de inicio de Django, por lo que solucionaremos esto.

Debemos definir una vista de inicio en el archivo `views.py` y asignarle una URL raíz en el archivo `urls.py` principal (del proyecto Django). Luego, creamos un template (plantilla) HTML para esa página y la colocamos en la carpeta `templates` de tu aplicación. Finalmente, asociamos esa plantilla con la vista en el archivo `urls.py` principal para que se cargue al acceder a la raíz de la aplicación.

Pasos para cargar una página por defecto en Django:

1. Crea una vista en `views.py`:

En el archivo `views.py` de tu aplicación, crea una función de vista que renderice tu página HTML. Por ejemplo:

```
# Create your views here.
```

```
def pagina_inicio(request):  
    return render(request, 'mi_aplicacion/inicio.html')
```

2. Crea la carpeta `templates` (si no existe):

Dentro de la carpeta raíz de tu aplicación crea una carpeta llamada `templates`.

3. Agrega la carpeta de `templates` a `settings.py`:

Asegúrate de que settings.py conozca la ubicación de tus plantillas añadiendo la siguiente línea dentro del apartado TEMPLATES:

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')], ←
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    ],
]
```

#### 4. Define la URL raíz en urls.py principal:

En el archivo urls.py de tu proyecto principal (proyecto Django), importa tu vista y asígnala a la URL raíz (""):

```
urlpatterns = [
    path('admin/', admin.site.urls),
    path('', views.pagina_inicio, name='home'), ←
    path('biblioteca/', include('biblioteca.urls'))
]
```

#### 5. Crea el archivo HTML en el directorio de plantillas:

Crea un archivo html dentro de mi\_aplicacion/templates/home/ (o la ruta que hayas especificado en views.py). Este será tu página de inicio por defecto.

# Create your views here.

```
def pagina_inicio(request):
    return render(request, 'mi_aplicacion/inicio.html')
```

En este caso, si definimos el archivo de nombre inicio.html, será el que debemos crear.

6. Creamos el contenido HTML de la página de inicio. Dentro de esta definiremos elementos de navegación y los que sean necesarios para trabajar inicialmente con nuestra aplicación.

Con estos pasos, cada vez que un usuario acceda a la URL principal de nuestro proyecto Django, se cargará la vista home y, en consecuencia, el archivo inicio.html que hayamos definido.

Hay 2 tipos de vistas a usar en Django, vistas basadas en funciones (FBV) como la que ocupamos para cargar la página inicial:

```
# Create your views here.
```

```
def pagina_inicio(request):  
    return render(request, 'mi_aplicacion/inicio.html')
```

Y también tenemos las vistas basadas en clases (CBV) que son vistas que usan una plantilla creada por Django, pero que tendrán como base de sus componentes a una clase específica.

```
class BibliotecaViewSet(viewsets.ModelViewSet):  
    queryset = Biblioteca.objects.all()  
    serializer_class = bibser
```

Cada uno de estos tipos de vistas nos permitirán establecer la lógica de datos que las acompañará, como en cualquier otro proyecto con patrón MVC (Modelo Vista Controlador)

## Modificar Vistas de Clase

El siguiente paso será modificar nuestras vistas de clase CBV (Class Based View).

1. Lo primero que debemos hacer es crear cada una de nuestras vistas basadas en clases:

```

class TipoCategoria_ViewSet(viewsets.ModelViewSet):
    queryset = TipoCategoria.objects.all()
    serializer_class = tipser

class Categoria_ViewSet(viewsets.ModelViewSet):
    queryset = Categoria.objects.all()
    serializer_class = catser

class Libro_ViewSet(viewsets.ModelViewSet):
    queryset = Libro.objects.all()
    serializer_class = libser

class Prestamo_ViewSet(viewsets.ModelViewSet):
    queryset = Prestamo.objects.all()
    serializer_class = preser

```

En los ejemplos mostrados tenemos CBV usando como base ModelViewSet, que proporciona un conjunto completo de operaciones CRUD (Crear, Leer, Actualizar, Eliminar) para un modelo.

Viene del framework REST de Django y automáticamente implementa los métodos necesarios para interactuar con los modelos de la base de datos, como obtener una lista de registros, crear uno nuevo o recuperar un registro individual por su ID.

Es ideal para construir APIs REST potentes y rápidas, ya que automatiza la creación de la lógica de la API para un modelo.

Estas vistas recibirán una solicitud web (un objeto HttpRequest) y devuelven una respuesta web (un objeto HttpResponse o una redirección). Luego, en el archivo urls.py, debemos mapear una URL a nuestra vista para que Django sepa cuándo llamar a la lógica para una solicitud HTTP.

## 2. Define las URL (en urls.py):

En el archivo urls.py de tu aplicación, mapea una URL a tu vista.

Ejemplo: (en mi\_aplicacion/urls.py):

```

urlpatterns = [
    path('', include(router.urls)),
    path('biblioteca/listado_nacionalidades',
         views.NacionalidadListView.as_view(), name='biblioteca')
]

```

3. Crea tus propias vistas, usando los conocimientos anteriores para navegar por la aplicación y a probar tus CBV.

Revisen la documentación directa del sitio en <https://www.django-rest-framework.org/>