



# **Servidor básico con Django**

Débora Rubio

**Enlace al repositorio:** [https://github.com/debsign/UE\\_MOD6\\_desarrollo\\_backend/tree/main/ejercicio\\_entregable\\_3](https://github.com/debsign/UE_MOD6_desarrollo_backend/tree/main/ejercicio_entregable_3)

## Explicación del código desarrollado

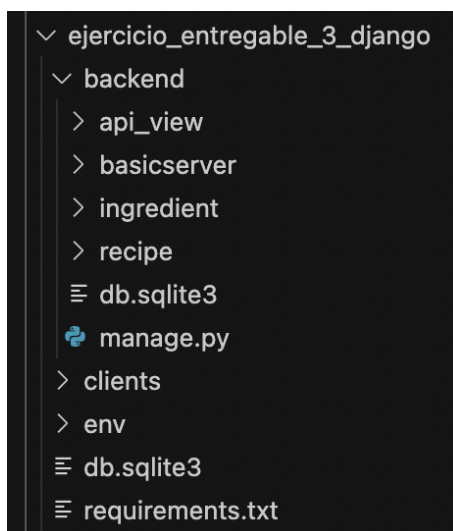
Estos son los pasos que hemos seguido para el desarrollo de un servidor con Django que implemente todas las herramientas vistas en la unidad 3 del módulo.

La temática elegida debe cumplir que el servidor tenga sentido, que los modelos tengan una relación lógica entre sí.

Los requisitos son los siguientes:

1. Generar al menos dos modelos distintos
2. Que sea portable (es decir, que exista un archivo `requirements.txt` que contenga todas las dependencias necesarias para su funcionamiento)
3. Aplicar DRY
4. Utilizar al menos 4 vistas genéricas distintas para cada modelo
5. Al menos una `api_view` propia que enlace modelos (que no sean vistas genéricas)

Esta es la estructura final de mi proyecto:



Por lo que iremos desgranando cada parte, según las hemos necesitado. Lo principal ha sido crear el entorno, junto con las carpetas **backend** y **clients**. Dentro de backend se encuentra nuestro servidor, la carpeta **api\_view** y las carpetas de cada modelo que vaya a crear, todo junto al archivo *manage.py*. En mi caso para los modelos he escogido la temática “receta” e “ingrediente”.

- ingredient

A destacar, lo que hemos ido creando en el primer modelo ha sido:

El archivo models.py, donde definimos los campos del modelo:

```
from django.db import models

# Create your models here.
class Ingredient(models.Model):
    name = models.CharField(max_length=100)
    quantity = models.IntegerField()
```

El archivo serializer.py correspondiente al modelo:

```
from rest_framework import serializers

from .models import Ingredient

class IngredientSerializer(serializers.ModelSerializer):
    class Meta:
        model = Ingredient
        fields = ('name', 'quantity')
```

El archivo views.py donde vamos a definir las acciones para con el modelo (muy similar a lo definido en las clases):

```
from .models import Ingredient
from .serializer import IngredientSerializer

from rest_framework import generics

class IngredientRetrieveAPIView(generics.RetrieveAPIView):
    queryset = Ingredient.objects.all()
    serializer_class = IngredientSerializer

class IngredientListCreateAPIView(generics.ListCreateAPIView):
    queryset = Ingredient.objects.all()
    serializer_class = IngredientSerializer

class IngredientUpdateAPIView(generics.UpdateAPIView):
    queryset = Ingredient.objects.all()
    serializer_class = IngredientSerializer
    def perform_update(self, serializer):
        instance = serializer.save()

class IngredientDestroyAPIView(generics.DestroyAPIView):
    queryset = Ingredient.objects.all()
    serializer_class = IngredientSerializer
    def perform_destroy(self, instance):
        super().perform_destroy(instance)
```

El archivo urls.py donde asignamos a cada vista una url:

```
from django.urls import path
from . import views

urlpatterns = [
    path('<int:pk>', views.IngredientRetrieveAPIView.as_view()),
    path('', views.IngredientListCreateAPIView.as_view()),
    path('<int:pk>/update/', views.IngredientUpdateAPIView.as_view()),
    path('<int:pk>/destroy/', views.IngredientDestroyAPIView.as_view()),
]
```

Y repetimos proceso para el modelo “recipe”, con la diferencia de que ahora necesitamos importar el modelo “ingredient” porque forma parte del modelo recipe:

- recipe

El archivo models.py, donde definimos los campos del modelo:

```
from django.db import models
from ingredient.models import Ingredient

class Recipe(models.Model):
    title = models.CharField(max_length=100)
    steps = models.TextField()
    ingredients = models.ManyToManyField(Ingredient)
```

Lo más importante de esta definición es la asignación que le hacemos a ingredients del modelo Ingredient utilizando ManyToManyField, donde se establece la relación de muchos a muchos entre las recetas y los ingredientes.

En el contexto de un sistema de recetas, una receta puede tener múltiples ingredientes, y a su vez, un ingrediente puede estar presente en varias recetas diferentes. Esta relación de muchos a muchos se representa utilizando un campo ManyToManyField en Django (por lo que hemos podido encontrar en la documentación y en ejemplos). Por lo que así queda definida la relación entre modelos, entre otras cosas.

El resto de archivos son muy similares a los vistos ya en el modelo ingredients, con la particularidad de que en el serializer.py de recipe también importamos el serializer.py de ingredient, porque el campo ingredients en el modelo recipe es precisamente (como ya hemos visto) una relación ManyToManyField con el modelo Ingredient:

```
from rest_framework import serializers
from ingredient.serializer import IngredientSerializer

from .models import Recipe

class RecipeSerializer(serializers.ModelSerializer):
    ingredients = IngredientSerializer(many=True, read_only=True)

    class Meta:
        model = Recipe
        fields = [
            'title',
            'steps',
            'ingredients',
        ]
```

Al importar el serializador de ingredient, se puede utilizar para incluir la representación de los ingredientes asociados a una receta. Esto permite que cuando se serializa una instancia de Recipe, también se incluyan los datos de los ingredientes relacionados.

También configuramos aquí el serializador de ingredientes para que pueda mostrar múltiples instancias de ingredientes y se establece como solo lectura (`read_only=True`). Esto hace que, al serializar una receta, se incluyan los datos de los ingredientes, pero de momento no permitimos actualizar directamente los ingredientes a través del serializador de recetas.

#### - api\_view

```
from recipe.models import Recipe
from ingredient.models import Ingredient
# rest_framework
from rest_framework.decorators import api_view
from rest_framework.response import Response

from recipe.serializer import RecipeSerializer
from ingredient.serializer import IngredientSerializer

@api_view(['GET'])
def api_home(request, *arg, **kwargs):
    instance = Recipe.objects.all().order_by('?').first()
    data = {}

    if instance:
        data = RecipeSerializer(instance).data

    return Response(
        data
    )

@api_view(['GET'])
def get_ingredient_recipes(request, *args, **kwargs):
    ingredients = Ingredient.objects.all()
    ingredient_data = IngredientSerializer(ingredients, many=True).data
    recipes = Recipe.objects.all()
    recipe_data = RecipeSerializer(recipes, many=True).data
    # Creamos un diccionario con los datos de ingredientes y recetas
    data = {
        'ingredients': ingredient_data,
        'recipes': recipe_data
    }

    return Response(data)
```

Aquí lo que hacemos es definir dos funciones decoradas con `@api_view(['GET'])` que actúan como vistas en el servidor.

La función `api_home` devuelve una receta seleccionada aleatoriamente (tal y como hicimos en clase), mientras que la función `get_ingredient_recipes` devuelve todos los ingredientes y todas las recetas disponibles en la base de datos (y es la función que integra los dos modelos, tal y como solicita el enunciado del ejercicio).

El archivo `urls.py` define la url de las funciones anteriores:

```
from django.urls import path
from . import views

urlpatterns = [
    path('', views.api_home),
    path('ingredient-recipes/', views.get_ingredient_recipes)
```

- `basic_server`

Para que todo esto funcione, debemos tener registrados nuestros modelos y la `api_view` en las `INSTALLED_APPS` del archivo `settings.py`.

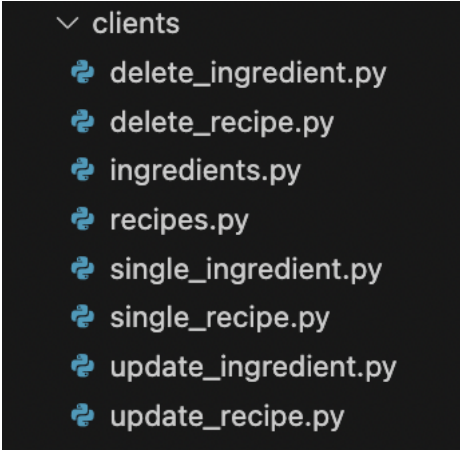
Además, en el archivo `urls.py` debemos definir todas las urls necesarias, tanto de los modelos como de la `api_view` (las dos que tenemos.)

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

urlpatterns = [
    path('admin/', admin.site.urls),
    # se incluyen las urls del archivo creado en api_view
    path('api_view/', include('api_view.urls')),
    path('api_view/recipe/', include('recipe.urls')),
    path('api_view/ingredient/', include('ingredient.urls')),
    path('api_view/ingredient-recipes/', include('api_view.urls')),
]
```

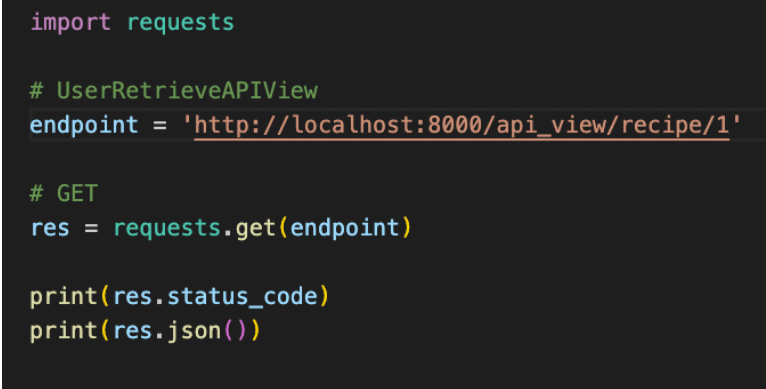
- clients

Hemos creado para cada una de las peticiones de las vistas un archivo, la estructura es la siguiente:



```
▼ clients
  delete_ingredient.py
  delete_recipe.py
  ingredients.py
  recipes.py
  single_ingredient.py
  single_recipe.py
  update_ingredient.py
  update_recipe.py
```

En `single_recipe.py`, por ejemplo, hacemos una solicitud GET a la URL `http://localhost:8000/api_view/recipe/1`. La parte `/api_view/recipe/1` de la URL nos dice que se está solicitando información sobre la receta con el `id=1`. Por lo tanto, el código enviará una solicitud GET a esa URL y mostrará (en este caso) el código de estado de la respuesta y el contenido JSON de la respuesta en consola.



```
import requests

# UserRetrieveAPIView
endpoint = 'http://localhost:8000/api_view/recipe/1'

# GET
res = requests.get(endpoint)

print(res.status_code)
print(res.json())
```

En otros casos donde nos ha costado un poco más hemos considerado controlar mejor los posibles errores que pudiésemos tener al realizar las consultas, como por ejemplo en `delete_recipe.py`:

```
import requests
import json

# RecipeDestroyAPIView
endpoint = 'http://localhost:8000/api_view/recipe/2/destroy/'

# DESTROY
res = requests.delete(endpoint)

print(res.status_code)

try:
    response_data = res.json()
    print(response_data)
except json.decoder.JSONDecodeError:
    if res.status_code == 204:
        print("Receta borrada.")
    else:
        print("Error")
```

En él, intentamos decodificar la respuesta JSON utilizando `res.json()`. Si la decodificación es correcta y no se genera ninguna excepción, se asume que la respuesta contiene datos JSON válidos y se imprime `response_data`. Pero, si se produce una excepción `JSONDecodeError` (nos lo ha proporcionado la documentación), se ejecutará el bloque `except` correspondiente.

En el bloque `except`, verificamos si el código de estado de la respuesta es 204 (No Content), lo cual indica que la solicitud DELETE se ha hecho correctamente y no hay contenido en la respuesta. En este caso, se imprimirá "Receta borrada.". Si el código de estado no es 204, se imprime "Error".

Y ahora comprobamos todo:

Abrimos la shell de Python mientras el servidor está corriendo y probamos a crear varios ingredientes, varias recetas y a asociarlos:

# Importamos modelos

```
from recipe.models import Recipe
from ingredient.models import Ingredient
```

# Creamos ingredientes

```
tomate = Ingredient.objects.create(name='Tomate', quantity=100)
queso = Ingredient.objects.create(name='Queso', quantity=50)
patata = Ingredient.objects.create(name='Patata', quantity=300)
```

# Creamos recetas



```
receta_1 = Recipe.objects.create(title='Ensalada de tomate y queso', steps='Paso 1: Cortar el tomate y el queso, Paso 2: Poner en un plato todo mezclado, Paso 3: Aliñar con aceite y especias')
```

```
receta_2 = Recipe.objects.create(title='Pizza de tomate y queso', steps='Paso 1: Comprar una masa de pizza congelada, Paso 2: Poner primero el tomate por encima, Paso 3: Espolvorear el queso por encima, Paso 4: Introducir 20 minutos en el horno a 180 grados, Paso 5: Sacar del horno')
```

```
receta_3 = Recipe.objects.create(title='Ensalada de patata y queso', steps='Paso 1: Cortar las patatas y el queso, Paso 2: Poner en un plato todo mezclado, Paso 3: Aliñar con aceite y especias')
```

```
# Asociamos ingredientes a recetas
```

```
receta_1.ingredients.add(tomate, queso)
receta_2.ingredients.add(tomate, queso)
receta_3.ingredients.add(patata, queso)
```

Como en el proceso nos hemos equivocado algunas veces, hemos visto necesario buscar y encontrar la forma de borrar todas las entradas de la base de datos para volver a empezar de cero de la siguiente forma:

```
# Obtenemos todas las recetas
```

```
recipes = Recipe.objects.all()
```

```
# Eliminamos todas las asociaciones de ingredientes en cada receta
```

```
for recipe in recipes:
```

```
    recipe.ingredients.clear()
```

```
# Borramos todos los registros de ingredient
```

```
Ingredient.objects.all().delete()
```

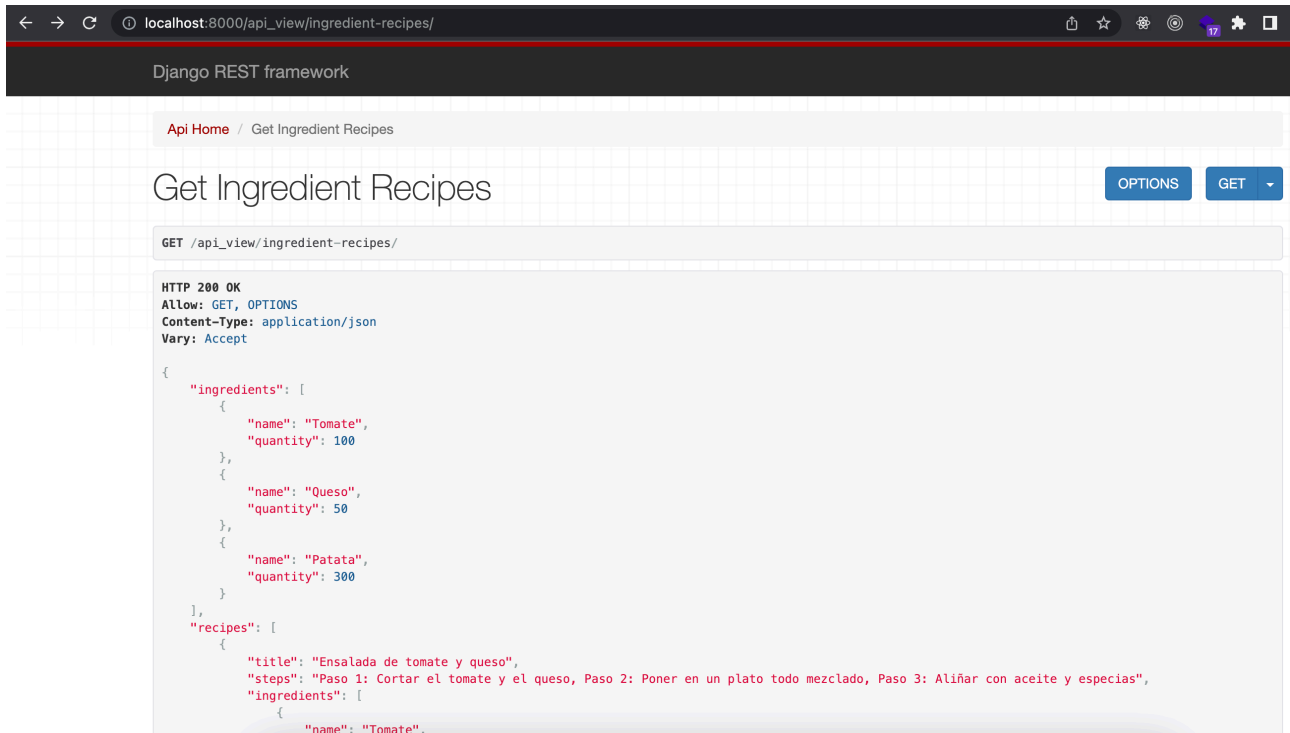
```
# Borramos todos los registros de recipe
```

```
Recipe.objects.all().delete()
```

Y en otra terminal comprobamos, estando en la carpeta clients, a ejecutar cada uno de los archivos, mostrándose los resultados, por ejemplo:

```
(env) deborarubiosoliva@MacBook-Air-de-Debora-2 clients %
python ingredients.py
Ingrediente 1:
{'name': 'Tomate', 'quantity': 100}
-----
Ingrediente 2:
{'name': 'Queso', 'quantity': 50}
-----
Ingrediente 3:
{'name': 'Patata', 'quantity': 300}
-----
```

También lo podemos comprobar en el propio navegador:



## Conclusiones y observaciones

El ejercicio ha sido muy interesante, pero personalmente quizá ha sido el más complicado para mí de todo el master, ya que mis nociones de backend son más escasas, pero para eso estamos aquí. El tema de controlar los espacios y carpetas, lo necesario para que todo funcione y demás me ha resultado complejo, especialmente al principio.

Pese al retraso en la fecha de entrega, mi situación personal me ha permitido dedicarle menos tiempo a este entregable, por lo que siento que podría estar más trabajado, sobre todo a la hora de controlar los errores en las peticiones en el cliente, e incluso crear más funciones con `api_view`.

Una de las cosas que más me ha perjudicado es la asignación de ids por parte de Django de forma automática, de forma que, aunque borrarse elementos, los ids ya estaban asignados y ocupados, me hubiese gustado saber cómo hacer para “reiniciar” de alguna manera estos ids (estoy segura que con haberlo preguntado en alguna tutoría se hubiese resuelto, pero ya ha sido tarde).

Creo que en los proyectos grupales y personal tendré más tiempo y dedicación a esta parte, gran motivación para mí a la hora de decidirme por este master, por lo que espero y deseo mejorar mis conocimientos sobre backend en estos meses, con esta breve (pero intensa) introducción a Django.