API REST CON DJANGO REST FRAMEWORK - Parte 1

RE Representational
S State
T Transfer
(Transferencia de estado representacional)

A Application
P Programming
I Interface

(Interfaz de programación de aplicaciones)

El concepto de REST (Representational State Transfer) es un estilo de arquitectura para diseñar servicios web, particularmente APIs, que permite la comunicación entre sistemas de forma sencilla, escalable y eficiente a través de protocolos como HTTP. A continuación, detallo los puntos clave para entender REST y su uso en APIs REST:

Qué es REST

REST no es un estándar o protocolo, sino un conjunto de principios arquitectónicos definidos por Roy Fielding en su tesis doctoral en el año 2000.

Está basado en la idea de recursos, que son entidades o datos que pueden ser representados y manipulados a través de operaciones específicas (para entendernos, los archivos u objetos que solicitamos cuando accedemos a una página web)

¿Cuáles son los Principios de REST?

Para que una API sea considerada RESTful, debe seguir estos principios básicos:

- **Identificación de recursos mediante URIs:** Cada recurso (entidad o dato) se identifica mediante un URI (Uniform Resource Identifier).

Por ejemplo:
  - `https://api.example.com/users/123` identifica al usuario con ID 123.
  - `https://api.example.com/products/456` identifica al producto con ID 456.

- **Uso de métodos HTTP:** REST aprovecha los métodos estándar de HTTP para realizar operaciones sobre los recursos:
  - GET: Recuperar un recurso (lectura).
  - POST: Crear un recurso nuevo.
  - PUT: Actualizar un recurso existente (reemplazo completo).
  - PATCH: Modificar parcialmente un recurso.
  - DELETE: Eliminar un recurso.

- **Representación de recursos:** Los recursos se pueden representar en diferentes formatos, como JSON (el más común), XML, HTML, o incluso texto plano.

- **Sin estado (stateless):** Cada solicitud al servidor debe ser independiente y no almacenar información de estado entre solicitudes. Toda la información necesaria para procesar una solicitud debe enviarse con esta (por ejemplo, en los encabezados o parámetros).

- **Comunicación mediante un cliente-servidor:** La arquitectura REST separa el cliente (que solicita recursos) y el servidor (que los provee). Esto permite mayor escalabilidad y flexibilidad en el desarrollo.

- **Cacheabilidad:** Las respuestas de los recursos deben indicar si se pueden almacenar en caché para mejorar el rendimiento y reducir la carga en el servidor.

- **Interfaz uniforme:** Todos los recursos deben exponerse de forma uniforme, lo que implica:
    - Uso consistente de URIs y métodos.
    - Representaciones estándar de datos.
    - Uso claro de códigos de estado HTTP.

Imagina un sistema de gestión de usuarios. Una API RESTful podría exponer los siguientes endpoints:

- GET /users: Recuperar la lista de usuarios.
- GET /users/123: Recuperar la información del usuario con ID 123.
- POST /users: Crear un nuevo usuario.
- PUT /users/123: Actualizar completamente el usuario con ID 123.
- PATCH /users/123: Actualizar parcialmente la información del usuario 123.
- DELETE /users/123: Eliminar al usuario con ID 123.

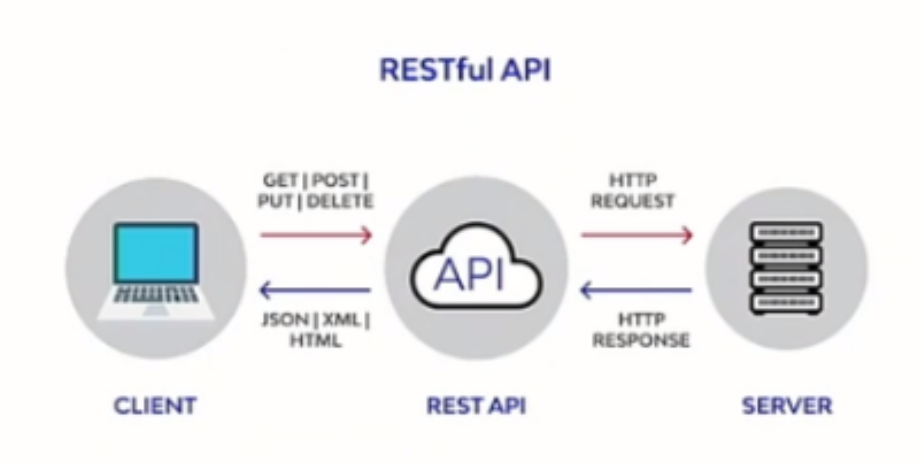
Ventajas de REST

- Escalabilidad: Separación cliente-servidor permite escalar sistemas con facilidad.
- Interoperabilidad: Soporte multiplataforma y diversos formatos de datos.
- Simplicidad: Aprovecha conceptos y estándares ampliamente conocidos como HTTP y JSON.
- Flexibilidad: Funciona bien con aplicaciones móviles, web, IoT y más.

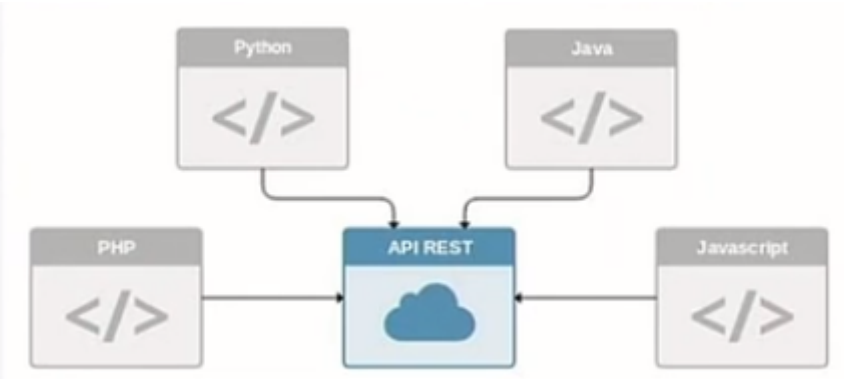
Diferencia con otros estilos

REST se diferencia de otros estilos, como SOAP, porque:
- Es más ligero (usa JSON en lugar de XML complejo).
- No depende de un protocolo rígido.
- Es más sencillo de implementar y entender.

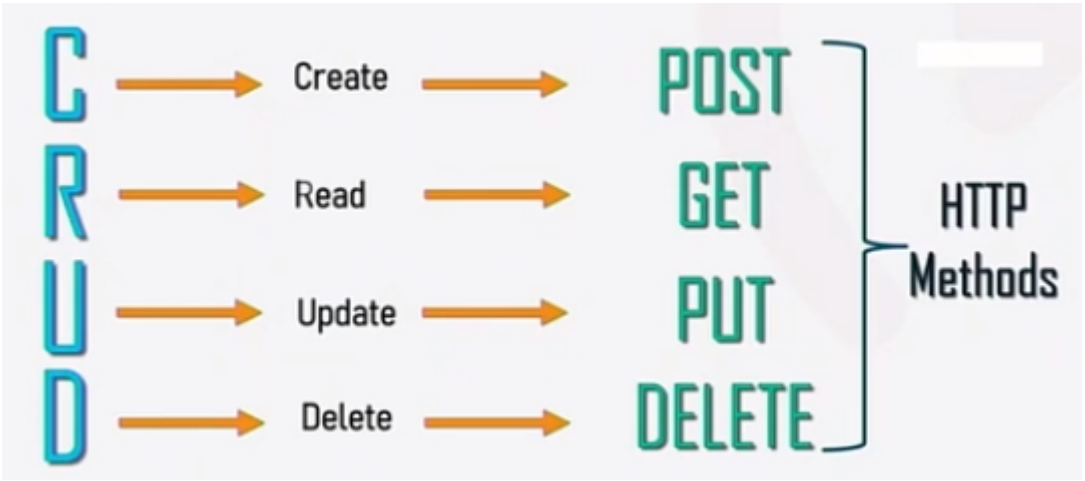
En resumen, una API REST es una interfaz que permite acceder y manipular recursos de manera eficiente, siguiendo los principios arquitectónicos descritos arriba.



Supongamos que el cliente (una aplicación) necesita información de las BD alojadas en el servidor; a través del servicio REST se utilizará una API para consultar esos datos y devolverlos en un formato que el cliente pueda entender y consumir. No hay acceso directo desde el cliente al servidor sino que es el servicio REST a través de la API el que sirve de intermediario.
Se utilizará un protocolo que el cliente debe conocer para poderle servir los datos solicitados.



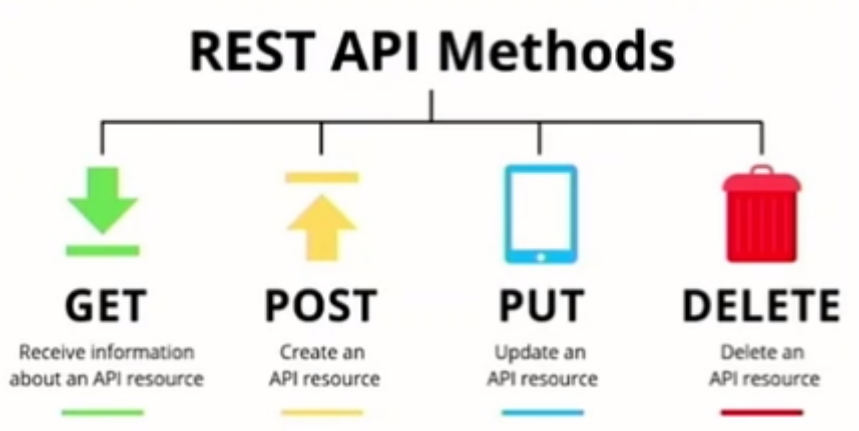
Las API no están ligadas a ningún lenguaje ni a ninguna plataforma. Generando una sola API los datos van a ser accesibles independientemente del lenguaje de programación de la aplicación cliente.



Las operaciones básicas que el cliente realiza con los datos que solicita son las CRUD. Para ello, se utilizarán una serie de métodos para realizar esas solicitudes que son los representados en la imagen.

HTTP es el protocolo que permite enviar documentos de un lado a otro en la web (es decir, es el protocolo que se utiliza para el intercambio de información entre cliente y servidor). Un protocolo es un conjunto de reglas que determina qué mensajes se pueden intercambiar y qué mensajes son respuestas apropiadas a otros.

Cada solicitud especifica un cierto método HTTP en el encabezado de la solicitud.
Los verbos HTTP le indican al servidor qué hacer con los datos identificados por la URL.



Nuestra API recibirá la misma información, pero dependiendo del método con el que se haga la petición decide si va a realizar una acción u otra. El método HTTP con el que nos conectamos con el servicio REST va a definir qué acciones estoy queriendo realizar.

CONCEPTO DE URI Y URL

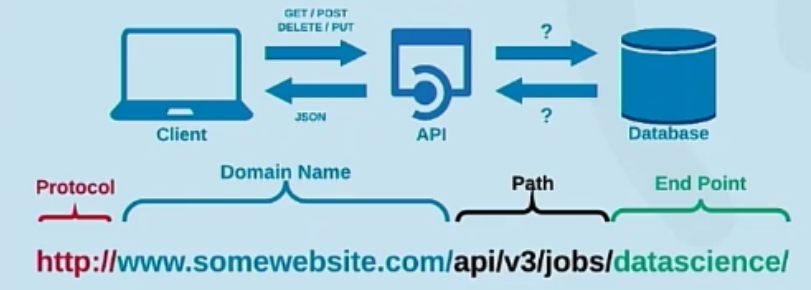
URI: Identificador uniforme de recursos (Uniform Resource Identifier). Se trata del identificador del recurso (una cadena de caracteres que identifica el recurso en la web y por lo tanto incluye su localización)

URL: Localizador de recursos uniforme (Uniform Resources Locator). Se trata de la cadena de caracteres que localiza un recurso en la web.

Toda URL es una URI (porque localiza un recurso en la web y lo identifica), pero no toda URI es una URL. La URL incluye información adicional, como el protocolo (http://) y el dominio, mientras que la URI se enfoca más en identificar el recurso.

URL completa: http://www.somewebsite.com/api/v3/jobs/datascience
URI del recurso: /api/v3/jobs/datascience

CONCEPTO DE END-POINT



Cuando se lanza una petición a un servicio REST, la petición se compone de varias partes:

- Protocolo
- Dominio
- Ruta
- End Point: **Define exactamente qué recurso se está solicitando.**

Es probable que haya varios end points idénticos pero diferenciándose en el método HTTP. Según el método empleado se ejecutará una acción u otra en ese end point. Por tanto es tan importante el end point (recurso solicitado) como el método con el que se está accediendo a él.
El formato que devuelve la API puede ser muy variado aunque lo más usados son XML y sobre todo JSON.

DJANGO REST FRAMEWORK

Django es un framework de desarrollo que se puede utilizar para desarrollar aplicaciones web de forma rápida y eficiente. La mayoría de las aplicaciones web tienen varias funciones comunes, como la autenticación, la recuperación de información de una base de datos y la administración de cookies. Los desarrolladores tienen que codificar una funcionalidad similar en cada aplicación web que escriban. Django facilita su trabajo al agrupar las diferentes funciones en una gran colección de módulos reutilizables, llamada marco de aplicación web. Los desarrolladores utilizan el marco

web de Django para organizar y escribir su código de manera más eficiente y reducir significativamente el tiempo de desarrollo web.

¿Cómo funciona Django?

Cualquier aplicación web consta de dos partes: código de servidor y código de cliente. El cliente o visitante del sitio web tiene un navegador. Cuando escriben una URL en su navegador, envían una solicitud a la máquina del servidor web en la que se ejecuta la aplicación web. El servidor procesa la solicitud mediante una base de datos y envía información al cliente como respuesta. El código de cliente muestra la información al visitante como una página web.

Django administra el código para este sistema de solicitud y respuesta mediante el uso de una arquitectura **Plantilla de Vista Modelo** **(MVT: model view template. Diferente al MVC – modelo vista controlador)**

Model

Los modelos de Django actúan como interfaz entre la base de datos y el código del servidor. Son la única fuente definitiva de información sobre sus datos. Estos modelos de datos contienen los campos y las operaciones esenciales que necesita para interactuar con la base de datos. Los modelos de Django convierten así las tablas de la base de datos en clases u objetos en el código Python. Esto se denomina “mapeo objeto-relacional”. Digamos que Django tiene su propio ORM similar al que hemos estudiado en la UT3.

Vista

En Django, las **vistas** son una parte fundamental del framework que se encarga de manejar las solicitudes del usuario y devolver las respuestas correspondientes. Las vistas son funciones o clases que procesan la lógica de la aplicación. Para entenderlo mejor:

- Una **vista** es una función (o clase) de Python que recibe una solicitud HTTP (representada por un objeto `HttpRequest`) y devuelve una respuesta HTTP (representada por un objeto `HttpResponse`).
- La respuesta puede ser cualquier cosa: texto, código HTML, un archivo, un JSON, una imagen, un código de error, etc.
- Cada vista tiene un propósito específico, y generalmente hay una vista para cada tipo de funcionalidad o página web.

Django tiene un sistema de mapeo o enrutamiento que conecta las URLs con las vistas (funciones) correspondientes. Este mapeo se define en el archivo `urls.py` del proyecto.

Supongamos que queremos ver una lista de todos sus empleados en un año específico. A groso modo, los pasos a dar serían configurar la ruta URL empleado/año y escribir la función de vista year_archive de Django correspondiente:

¿qué contendría el archivo `urls.py`?:

```
from django.urls import path
from . import views  # Importamos las vistas desde el archivo views.py

urlpatterns = [
```

```
    path('employee/name', views.employee_name),   # Mapeo de una URL estática
    path('employee/<int:year>/', views.year_archive),  # Mapeo de una URL dinámica
]
```

- `path()`: Es una función que conecta una ruta URL específica con una función o clase de vista.
  - La primera parte (por ejemplo, `'employee/name'`) es la ruta de la URL.
  - La segunda parte (`views.employee_name`) es la función de vista que se ejecutará cuando alguien acceda a esa URL.
- También se pueden usar rutas dinámicas (como `<int:year>`) para capturar valores específicos de la URL y pasarlos a la vista como parámetros.

Cuando el visitante escribe “nombredelsitioweb.com/empleado/2020” en su navegador, se producen los siguientes pasos

1. **El visitante accede a una URL específica:**
   - Por ejemplo: `nombredelsitioweb.com/employee/2020`.
2. **Django procesa la solicitud:**
   - El sistema de enrutamiento busca en `urlpatterns` del archivo `urls.py` para encontrar una ruta que coincida con la URL solicitada.
   - Encuentra la ruta `'employee/<int:year>/'` y asocia la solicitud con la función de vista `year_archive`.
3. **La vista procesa la lógica:**
   - La función `year_archive` recibe el valor `2020` del parámetro dinámico `<int:year>`.
   - Utiliza un modelo de Django (por ejemplo, un modelo llamado `Employee`) para buscar todos los empleados registrados en el año 2020 en la base de datos.
   - Recopila estos datos en una estructura que se pueda devolver, como un diccionario o una lista.
4. **Django responde al navegador:**
   - La función de vista crea y devuelve una respuesta HTTP, como una página HTML con la lista de empleados, un JSON con los datos, etc.
   - El navegador del usuario muestra esta respuesta.

---

4. Ejemplo práctico

Aquí está el código de un ejemplo completo:

Archivo `urls.py`:

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

urlpatterns = [
    path('employee/<int:year>/', views.year_archive, name='year_archive'),
]
```

Archivo `views.py`:

```
from django.shortcuts import render
from .models import Employee  # Importamos el modelo Employee

def year_archive(request, year):
    # Consultamos empleados según el año proporcionado
    employees = Employee.objects.filter(hire_date__year=year)
    # Devolvemos una respuesta renderizando una plantilla HTML
     return render(request, 'employee/year_archive.html', {'year': year,
'employees': employees})
```

Plantilla `employee/year_archive.html`:

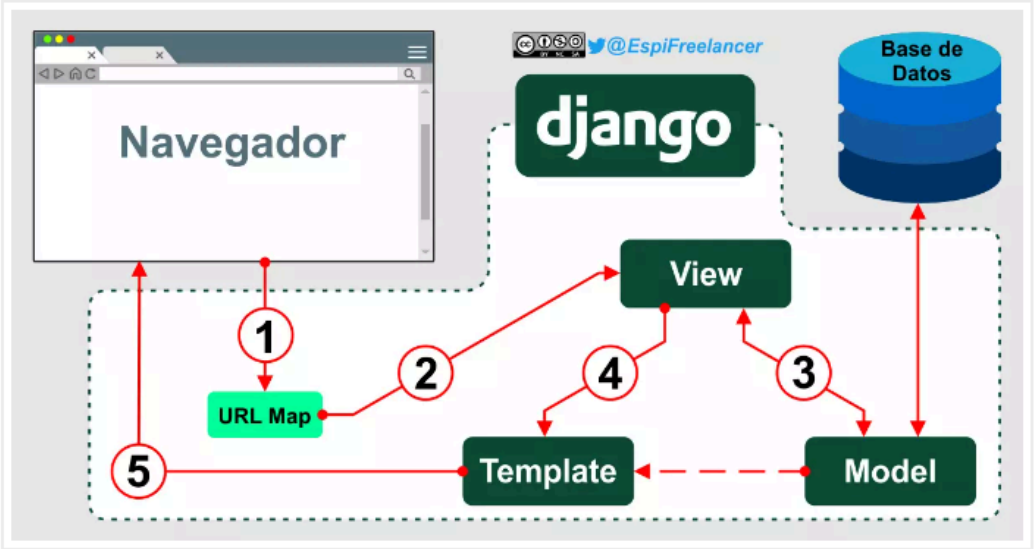
```
<!DOCTYPE html>
<html>
<head>
    <title>Empleados del año {{ year }}</title>
</head>
<body>
    <h1>Lista de empleados para el año {{ year }}</h1>
    <ul>
        {% for employee in employees %}
            <li>{{ employee.name }} - Contratado el {{ employee.hire_date
}}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Template

Las plantillas de Django administran la presentación de la página web en el navegador. Dado que la mayoría de las páginas web están en HTML, se puede escribir el código de la plantilla de Django en un estilo similar a HTML. Un archivo de plantilla contiene ciertos componentes:

- Las partes estáticas de la salida HTML final, como imágenes, botones y encabezados.
- Sintaxis especial que describe cómo insertar contenido o datos dinámicos, que cambia con cada solicitud.

Funcionamiento MTV de Django



Cuando se hace click en un enlace o se escribe una dirección (1) a lo primero que se accede es al mapa de URLs (también conocido como URL map o URL conf), en este archivo cada ruta está asociado con una view (2), si se necesita algún dato se solicitará a model (3) el cual a su vez generará la consulta a la base de datos, cuando los datos han sido traídos estos son enviados al template (4) que contiene la lógica de presentación para ellos. Después de "pintar" la página, esta se envía al navegador que hizo la solicitud (5).

DJango Rest Framework es un módulo que contiene un conjunto de herramientas potente y flexible para crear API web.

INSTALACIÓN Y PREPARACIÓN DEL PROYECTO:

1.- Crear una carpeta para el proyecto.
2.- Crear y activar un entorno virtual

Para evitar problemas, antes de comenzar a hacer ninguna configuración vamos a actualizar a la última versión de pip:

```
(.env)tu_ruta> python -m pip install --upgrade pip
```

3.- Ejecutar el fichero django.bat con los dos parámetros requeridos.

(.\django.bat 'nombre_proyecto' 'nombre_app')

%1 . → Nombre del proyecto (el que quieras. En mi caso en la imagen "DRF") Recuerda poner el punto después del nombre del proyecto y un espacio.
%2 → Nombre de la aplicación (el que quieras. En mi caso en la imagen "api")

- En el punto de la instalación del administrador indicar un password (recuerda que puede parecer que no estás escribiendo, pero sí).

- Actualizar DRF→**settings.py** dando de alta las aplicaciones recién instaladas (‘rest_framework’, ‘api’)

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

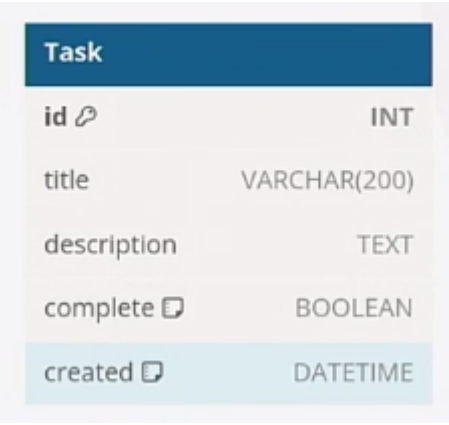
- Añadir en DRF→**urls.py** las rutas para poder tener acceso a las apis de Django
    - path(‘api-auth/’, include (‘rest_framework.urls’)),]

  En este punto si da error en el include importarlo de la librería Django (from Django import include)

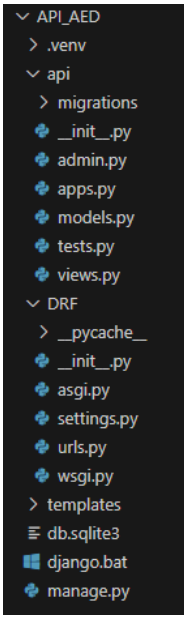
```
urls.py > ...
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
]
```

CREACIÓN DE UNA API PARA LOS MODELOS EXISTENTES

Crearemos una pequeña app “Lista ToDo” sobre la que podremos realizar CRUD.



Tras la ejecución de las instrucciones del apartado anterior nos quedará la siguiente estructura de carpetas:



Aprovecharemos el framework Django para crear la estructura de la base de datos, es decir, el modelo.
En el fichero **models.py** añadiremos el siguiente código:

```python
models.py > ...
from django.db import models

class Task(models.Model):
    title = models.CharField (max_length = 200, verbose_name="Título")
    description = models.TextField (null=True, blank = True, verbose_name="Descripción")
    complete = models.BooleanField (default = False, verbose_name = "Completada")
    created = models.DateTimeField (auto_now_add = True)

    def __str__(self) :
        return self.title
```

Y ahora debemos añadir a nuestro panel de administración el modelo que hemos creado. Esto se consigue añadiendo en el fichero **admin.py** las líneas 2 y 3 según se muestra en la imagen.

```python
admin.py
from django.contrib import admin
from .models import Task

admin.site.register (Task)
```

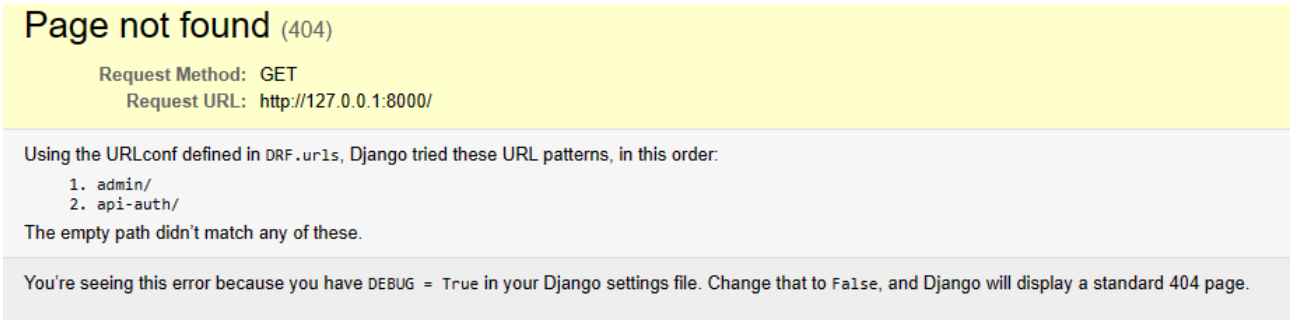
En este punto ya deberíamos tener acceso a la tabla creada, no obstante no se ha creado todavía. Debe hacerse primero una “migración”. Con el entorno virtual activado ejecutamos el comando ***python manage.py makemigrations,*** vemos que migración está preparada y a continuación ejecutamos la migración con el comando ***python manage.py migrate***

```
(.venv) PS D:\CURSO 22 - 23\Acceso a datos\API REST\API_AED> python manage.py makemigrations
Migrations for 'api':
  api\migrations\0001_initial.py
    - Create model Task
(.venv) PS D:\CURSO 22 - 23\Acceso a datos\API REST\API_AED> python manage.py migrate
Operations to perform:
  Apply all migrations: admin, api, auth, contenttypes, sessions
Running migrations:
  Applying api.0001_initial... OK
```

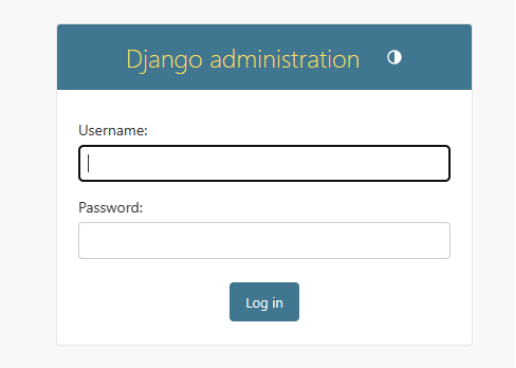
Arrancamos el servidor: ***python manage.py runserver***

```
(.venv) PS D:\CURSO 22 - 23\Acceso a datos\API REST\API_AED> python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
January 14, 2024 - 16:39:34
Django version 5.0.1, using settings 'DRF.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

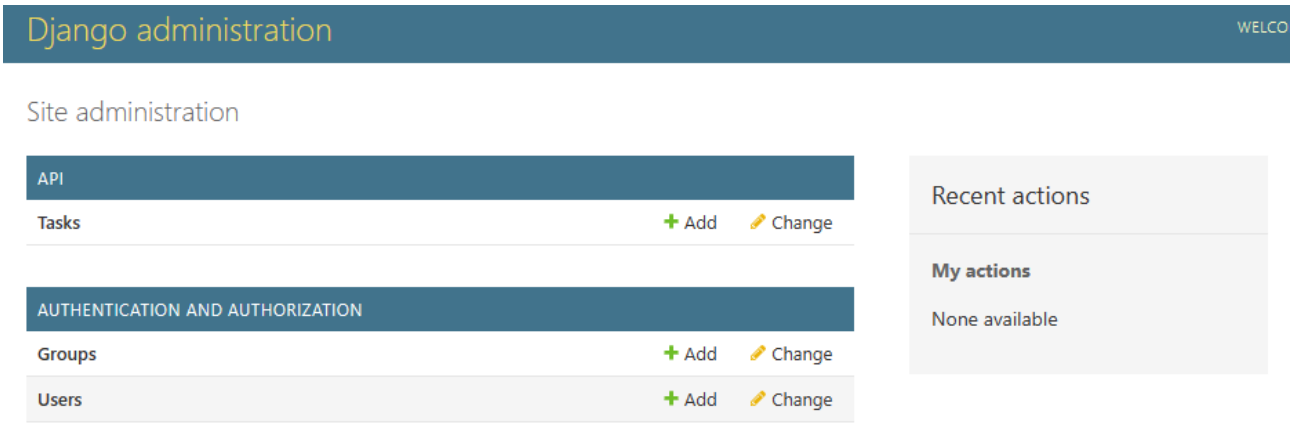
Puedes probar el funcionamiento abriendo el navegador y accediendo a 127.0.0.1:8000



Sabemos que está funcionando aunque no hemos especificado ninguna ruta válida por lo que nos arroja el error 404. Podemos escribir 127.0.0.1:8000/admin



Usando las credenciales indicadas cuando se instaló el administrador, en mi caso admin / admin se accede al panel de administración.



CREACIÓN DE LAS VISTAS

Inicialmente crearemos de forma manual nuestra API para entender mejor su funcionamiento interno y después utilizaremos el rest framework Django para ver cómo se facilita todo el trabajo realizado.

En el fichero **views.py** crearemos la clase TaskView a la que le definiremos 4 métodos:

```python
views.py > ...
from django.shortcuts import render
from django.views import View

class TaskView (View):
    def get (self, request):
        pass
    def post (self, request):
        pass
    def put (self, request):
        pass
    def delete (self, request):
        pass
```

Una vez creadas las vistas añadimos una nueva ruta del proyecto en el fichero **DRF/urls.py** para que se reconozcan todos los recursos que vamos a utilizar en nuestra API (3ª línea). Estos son los llamados end points. Previamente creamos el fichero ***urls.py*** dentro de la carpeta ***api (el fichero del mismo nombre existente está dentro de la carpeta DRF, pero necesitamos otro dentro de api)*** y metemos una configuración vacía de momento como se muestra en la imagen 2ª.

https://tutorialesinformatica.com/programacion/urls-en-django/?utm_content=cmp-true

En esta web se explica de manera clara el concepto y configuración de URLs en Django.

urls.py > ...

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api-auth/', include('rest_framework.urls')),
    path ('api/', include ('api.urls')),
]
```

api > urls.py > ...

```python
from django.urls import path, include

urlpatterns = [

]
```

IMPLEMENTACIÓN DE GET

Modificamos el código básico generado con las 4 vistas para añadir funcionalidad a la vista GET.

Observa las nuevas librerías que se han importado (HttpResponse, JsonResponse, Task) atendiendo a los comentarios incluidos en el código.

api > views.py > TaskView > get

```python
from django.shortcuts import render
from django.views import View
from django.http import HttpResponse, JsonResponse
#como vamos a trabar con el modelo creado debemos importarlo
from .models import Task

class TaskView (View):
    #Esta vista comprueba si hay alguna tarea. Si los hay los muestra y si no, lanza un mensaje.
    def get (self, request):
        #convertimos a listas para que los datos obtenidos se puedan serializa y convertir directamente a JSON.
        #podríamos haber escrito tasks = Task.objects.all(), pero esta fórmula nos generaría el error comentado es decir,
        #nos diría que los datos no son serializables y por tanto al intentar convertirlos a JSON daría error.
        tasks = list(Task.objects.values())
        if tasks:
            datos = {'message': 'OK!', 'tasks': tasks}
        else:
            datos = {'message': 'No hay tareas que hacer'}
        return JsonResponse(datos)
    #los datos se devuelven en formato JSON
```

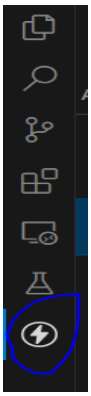
Cada vista debe llevar asociada una ruta que debemos incluir en el fichero **api/urls.py**

```
api > urls.py > ...
1   from django.urls import path, include
2   from .views import TaskView
3
4   urlpatterns = [path ('tasks/', TaskView.as_view(), name='task_list'),
5
6   ]
```

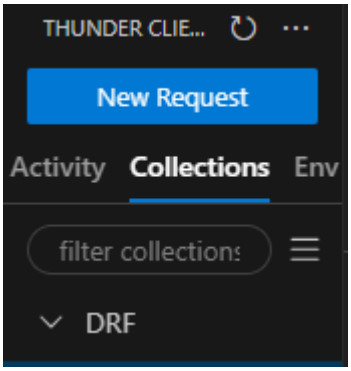
Con esta ruta creada, en el navegador bastaría con poner 127.0.0.1:8000/api/tasks para poder ver los registros del modelo en el formato que va a devolver la API.

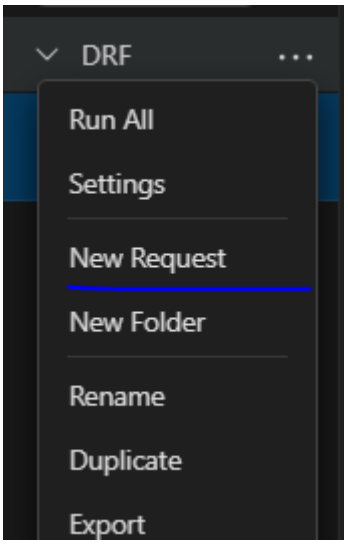
USO DEL CLIENTE THUNDER CLIENT PARA CONSUMO DE APIS

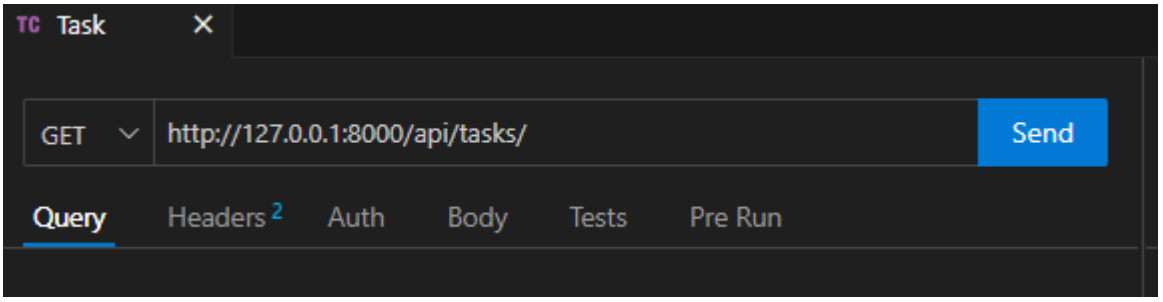
Instalamos en Visual Code la extensión Thunder Client. Aparecerá en la barra lateral izquierda un icono como se muestra en la imagen:

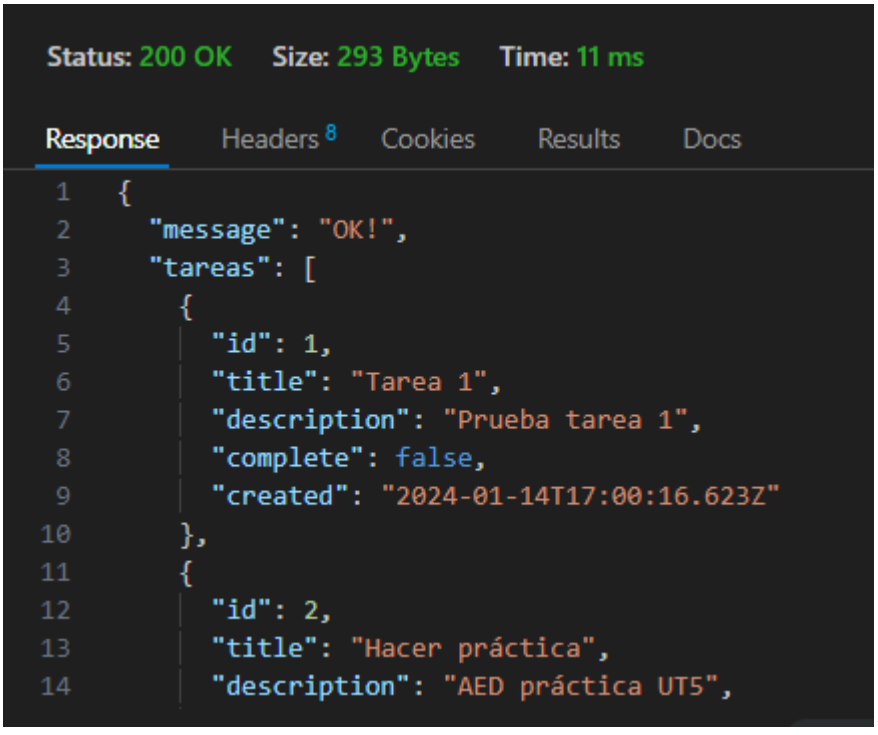


Iremos a la pestaña “Collections” y añadiremos una nueva (pulsando sobre las 3 líneas horizontales. En mi caso la he llamado “DRF”). Es posible que tengas varias y quieras diferenciar unas de otras, por ello crearemos una colección por cada api que vayamos a probar. Una vez creada, iniciamos un nuevo request. Se abrirá un cuadro de diálogo para escribir el nombre del request (escribe lo que quieras) y aparecerá el formulario para escribir el método y la URL que habíamos definido en nuestra view.


THUNDER CLIE...
New Request
Activity
Collections
Env
filter collections
DRF


DRF
Run All
Settings
New Request
New Folder
Rename
Duplicate
Export


TC Task
GET
http://127.0.0.1:8000/api/tasks/
Send
Query
Headers 2
Auth
Body
Tests
Pre Run


Status: 200 OK
Size: 293 Bytes
Time: 11 ms
Response
Headers 8
Cookies
Results
Docs
{
"message": "OK!",
"tareas": [
{
"id": 1,
"title": "Tarea 1",
"description": "Prueba tarea 1",
"complete": false,
"created": "2024-01-14T17:00:16.623Z"
},
{
"id": 2,
"title": "Hacer práctica",
"description": "AED práctica UT5",