

**DESARROLLO EN ENTORNO CLIENTE/SERVIDOR**

**UD10: SERVICIOS WEB Y APLICACIONES WEB**

**REACTIVAS**



**UD10 - Servicios web y aplicaciones web reactivas**

1. INTRODUCCIÓN.....	4
1.1. Request.....	4
1.2. Response.....	4
2. EVALUACIÓN.....	5
3. LA LIBRERÍA REQUESTS DE PYTHON.....	5
3.1. Consulta de datos.....	5
3.2. Utilización de parámetros de consulta.....	6
3.3. Creación de datos.....	7
3.4. Modificación de datos.....	7
3.5. Acceso a las cabeceras.....	8
3.6. Autenticación.....	8
3.6.1. Autenticación Basic.....	8
3.6.2. Autenticación Digest.....	9
3.6.3. Autenticación mediante sesión.....	9
3.6.4. Autenticación OAuth.....	9
3.7. Gestión de errores.....	13
3.8. Descarga de HTML.....	14
4. DJANGO REST FRAMEWORK.....	15
4.1. Instalación.....	16
4.2. Organización de ficheros.....	17
4.3. Serializadores.....	17
4.3.1. Categorías.....	18
4.3.2. Listado de proyectos.....	19
4.3.3. Detalle de proyecto.....	23
4.4. Vistas y Mixins.....	25

---

4.4.1. Lista de categorías.....	26
4.4.2. Mantenimiento de categorías.....	27
4.4.3. Lista de proyectos.....	30
4.4.4. Mantenimiento de proyectos.....	30
4.4.5. Métodos de las vistas.....	32
4.4.6. Validaciones.....	33
4.4.7. Parámetros.....	34
4.4.8. Ordenación.....	35
4.4.9. Búsqueda.....	37
4.4.10. Paginación.....	39
4.4.11. Vistas sin serializador.....	41
4.5. API navegable de DRF: test y documentación.....	43

## 1. INTRODUCCIÓN

Un **API** es un servicio que proporciona acceso a datos y métodos que otras aplicaciones pueden utilizar, mediante el protocolo HTTP, lo cual permite que pueda ser integrado en gran variedad de aplicaciones.

La variante de API más popular para servicios web actuales se denomina **REpresentational State Transfer** (REST), o también Servicio REST o API REST. Consiste en una serie de directrices diseñadas para simplificar la comunicación entre cliente y servidor.

### 1.1. Request

La forma de interactuar con un servicio REST es mediante un objeto llamado petición, solicitud o request, que está formada por:

- **Endpoint:** se trata de una URL que caracteriza el tipo de servicio (dato o acción) con el que estamos interactuando, dentro de un API.
- **Método:** especifica cómo interactuar con el endpoint. Los métodos más usuales son:
  - GET: Consulta de datos
  - PUT: Modificación de datos
  - POST: Creación de datos
  - DELETE: Eliminación de datos
- **Datos:** En aquellos métodos que necesiten cambio en los datos del sistema, se necesitará incluir en el request los datos a crear/modificar.
- **Cabeceras:** Contiene los metadatos adicionales necesarios, como tokens de autenticación, tipo de contenido a devolver en la respuesta, políticas de cacheado...

### 1.2. Response

Cuando se hace una request a un servicio REST, obtenemos una **respuesta** (o response), que al igual que en el caso del request, contendrá una cabecera y datos (cuando corresponda). Los datos retornados, en forma de texto, estarán normalmente en formato JSON (aunque es también posible utilizar otros formatos como XML).

## 2. EVALUACIÓN

El presente documento, junto con sus correspondiente boletín de actividades (publicado adicionalmente), cubren los siguientes criterios de evaluación:

RESULTADOS DE APRENDIZAJE	CRITERIOS DE EVALUACIÓN
RA7. Desarrolla servicios web reutilizables y accesibles mediante protocolos web, verificando su funcionamiento.	a) Se han reconocido las características propias y el ámbito de aplicación de los servicios web. b) Se han reconocido las ventajas de utilizar servicios web para proporcionar acceso a funcionalidades incorporadas a la lógica de negocio de una aplicación. c) Se han identificado las tecnologías y los protocolos implicados en el consumo de servicios web. d) Se han utilizado los estándares y arquitecturas más difundidos e implicados en el desarrollo de servicios web. e) Se ha programado un servicio web. f) Se ha verificado el funcionamiento del servicio web. h) Se ha documentado un servicio web.

## 3. LA LIBRERÍA REQUESTS DE PYTHON

La librería requests de python implementa los métodos HTTP que nos va a permitir interactuar con cualquier API REST. El primer paso es instalar la librería requests mediante el comando pip en un entorno virtual:

[pip install requests](#)

A continuación hemos de importar la librería en el script que queremos ejecutar:

```
import requests
```

En los siguientes subapartados se muestran las operaciones más básicas que se pueden realizar con esta librería. Vamos a utilizar varios REST API abiertos para poder hacer pruebas:

- [api.open-notify.org](http://api.open-notify.org): Utilizaremos este API REST para realizar varias peticiones de consulta.
- [httpbin.org](http://httpbin.org): Este API REST nos servirá para realizar peticiones de modificación/creación de datos, autenticación, entre otros.

### 3.1. Consulta de datos

Para hacer una consulta de datos a un endpoint que lo permita, utilizamos el método "get" de la librería requests, de la forma:

```
response = requests.get("http://api.open-notify.org/astros.json")
```

NOTA: Este API REST pretende proporcionar información sobre las personas que se encuentran actualmente en el espacio, no es una fuente oficial pero nos servirá de ejemplo.

Para visualizar el objeto returned, hacemos un print de response:

```
Python 3.8.10 (default, Sep 28 2021, 16:10:42)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import requests
>>> response = requests.get("http://api.open-notify.org/astros.json")
>>> print(response)
<Response [200]>
>>> █
```

Si queremos ver los atributos del objeto response, podemos utilizar la función dir():

```
>>> dir(response)
['__attrs__', '__bool__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__enter__', '__eq__', '__exit__', '__format__', '__ge__', '__getattr__', '__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__nonzero__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_content', '_content_consumed', '_next', 'apparent_encoding', 'close', 'connection', 'content', 'cookies', 'elapsed', 'encoding', 'headers', 'history', 'is_permanent_redirect', 'is_redirect', 'iter_content', 'iter_lines', 'json', 'links', 'next', 'ok', 'raise_for_status', 'raw', 'reason', 'request', 'status_code', 'text', 'url']
```

Para acceder al contenido de la respuesta, se pueden utilizar las siguientes opciones:

```
response.content() #respuesta en bruto (bytes)
response.text() #formato texto
response.json() #formato json
```

### 3.2. Utilización de parámetros de consulta

Si el endpoint lo permite, se pueden enviar parámetros de consulta en la petición "get" al servidor, con el fin de acotar los resultados.

Esto se puede llevar a cabo mediante el argumento "params" del método get. Este argumento ha de ser un diccionario con el parámetro/s a enviar al servidor. A continuación se muestra un ejemplo de consulta, para saber cuándo la Estación Espacial Internacional pasará por un determinado punto:

```
>>> query = {'lat':'60', 'lon':'80'}
>>> response = requests.get('http://api.open-notify.org/iss-pass.json', params=query)
>>> print(response.json())
{'message': 'success', 'request': {'altitude': 100, 'datetime': 1636646118, 'latitude': 60.0, 'longitude': 80.0, 'passes': 5}, 'response': [{'duration': 349, 'risetime': 1636646143}, {'duration': 555, 'risetime': 1636651777}, {'duration': 601, 'risetime': 1636657516}, {'duration': 579, 'risetime': 1636663292}, {'duration': 451, 'risetime': 1636669104}]}]
```

### 3.3. Creación de datos

Mediante httpbin.org es posible realizar peticiones tipo "post", aunque los datos no se almacenen realmente en una base de datos. Un ejemplo es el siguiente, donde se pasa al parámetro "data" un diccionario con los datos a crear:

```
response = requests.post('https://httpbin.org/post', data = {'valor':'nuevo'})
```

El resultado se puede ver en la siguiente captura:

```
>>> import requests
>>> response = requests.post('https://httpbin.org/post', data = {'valor':'nuevo'})
>>> print(response.text)
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "valor": "nuevo"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "11",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.26.0",
    "X-Amzn-Trace-Id": "Root=1-618d48e6-5831963d0a4583db1d9faa22"
  },
  "json": null,
  "origin": "213.0.87.154",
  "url": "https://httpbin.org/post"
}
```

### 3.4. Modificación de datos

De modo similar al método post, se puede realizar una llamada mediante put, para la modificación de datos:

```
response = requests.put('https://httpbin.org/put', data = {'valor':'modificado'})
```

```
>>> response = requests.put('https://httpbin.org/put', data = {'valor':'modificado'})
>>> print(response.text)
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "valor": "modificado"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Content-Length": "16",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.26.0",
    "X-Amzn-Trace-Id": "Root=1-618d499d-0945fb9b35e1b245687f43ca"
  },
  "json": null,
  "origin": "213.0.87.154",
  "url": "https://httpbin.org/put"
}
```

### 3.5. Acceso a las cabeceras

Para obtener los metadatos contenidos en la cabecera de la respuesta se utiliza el atributo headers del objeto response. Por ejemplo, para obtener la fecha y hora en que se produjo la respuesta:

```
print(response.headers["date"])
```

### 3.6. Autenticación

Para interactuar con servicios REST que necesitan una autenticación previa se pueden utilizar varios métodos, dependiendo del tipo de autenticación que requiera el endpoint. A continuación veremos la autenticación Basic y la basada en OAuth.

#### 3.6.1. Autenticación Basic

Es el método más simple y consiste en enviar un nombre de usuario y un password, equivalente a introducir usuario y password en un sitio web. En el ejemplo siguiente, se trata de obtener datos de la cuenta del usuario "user", y para ello se necesita enviar en la propia consulta los parámetros de autenticación para evitar un error de consulta no autorizada:

```
from requests.auth import HTTPBasicAuth
response = requests.get(
  'https://api.github.com/user',
```

```
    auth=HTTPBasicAuth('user', 'password')
)
```

O también, se puede realizar mediante una tupla:

```
requests.get('https://api.github.com/user', auth=('user', 'password'))
```

NOTA: En el caso de la API de Github, el valor de password se refiere al token personal de acceso (este punto se desarrollará en las actividades).

### 3.6.2. Autenticación Digest

Otro método común de autenticación es mediante Digest, de la forma:

```
from requests.auth import HTTPDigestAuth
url = 'https://httpbin.org/digest-auth/auth/user/pass'
response = requests.get(url, auth=HTTPDigestAuth('user', 'pass'))
```

### 3.6.3. Autenticación mediante sesión

La librería requests proporciona un objeto Session que permite conservar determinados parámetros para peticiones HTTP diferentes, así como cookies. Esto permite simular una sesión como si de un navegador web se tratase.

Un ejemplo de código puede ser el siguiente, en el que recuperamos los repositorios de un determinado usuario que se autentica previamente:

```
username = 'user'
token = 'XXX_token'
repos_url = 'https://api.github.com/user/repos'
gh_session = requests.Session()
gh_session.auth = (username, token)
print(gh_session.get(repos_url).text)
```

### 3.6.4. Autenticación OAuth

Existen dos estándares OAuth: OAuth1 y OAuth2. En este [enlace](#) puedes encontrar las diferencias fundamentales entre los dos métodos.

Para OAuth2, un ejemplo de autenticación mediante **Google** se detalla en esta [página](#). Según estas instrucciones, el funcionamiento básico sería:

1. A modo de ejemplo, se muestra a continuación una API de acceso a un proyecto de **Firebase**:

The screenshot shows the Google Cloud Platform API & servicios interface. On the left, a sidebar lists options like Panel, Biblioteca, Credenciales (which is selected), Pantalla de consentimiento ..., Verificación del dominio, and Acuerdos de uso de páginas. The main area is titled "ID de cliente para Aplicación web". It shows a form with a "Nombre \*" field containing "Web client (auto created by Google Service)". Below it, a note says: "El nombre de tu cliente de OAuth 2.0. Este nombre solo se usa para identificar al cliente en la consola y no se mostrará a los usuarios finales." To the right, there's a table with columns "ID de cliente", "Secreto del cliente", and "Fecha de creación". The "ID de cliente" row contains "481769355261-...". The "Secreto del cliente" row contains "luy...". The "Fecha de creación" row contains "12 de enero de 2017, 17:19:12 GMT+1". A red box highlights the "ID de cliente" and "Secreto del cliente" rows.

2. Luego, la aplicación cliente (la que estamos desarrollando) solicita un **token** de acceso del servidor de autorización de Google, extrae un token de la respuesta y envía el token a la API de Google a la que se desea acceder. El **scope**, que se envía como parámetro en esta solicitud de acceso, determina el conjunto de recursos y operaciones (dentro de la aplicación) a los que se puede acceder mediante el token. Es posible solicitar un scope, pero que se otorgue uno diferente, luego hay que comprobar el scope recibido.

3. En el acceso a una API, se adjunta el token de acceso en el **encabezado** de la solicitud HTTP. Los tokens de acceso son válidos solo para el conjunto de las operaciones y los recursos descritos en el scope de la solicitud de token. Por ejemplo, si se emite un token de acceso para la API de Google Calendar, no otorga acceso a la API de Contactos de Google. Sin embargo, puede enviar ese token de acceso a la API de Google Calendar varias veces para operaciones similares.

4. Los tokens otorgados tienen un tiempo de vida determinado, tras lo cual es necesario solicitar uno nuevo. Mediante el llamado **token de actualización**, que es un token proporcionado en el paso 2, diferente al token de acceso, se consigue obtener nuevos tokens de acceso una vez caducados mediante una solicitud específica de refresco.

NOTA: Existen diferentes variantes en el flujo de autenticación de OAuth2, dependiendo del tipo de aplicación que estemos desarrollando. En este [enlace](#) se pueden consultar diferentes flujos de

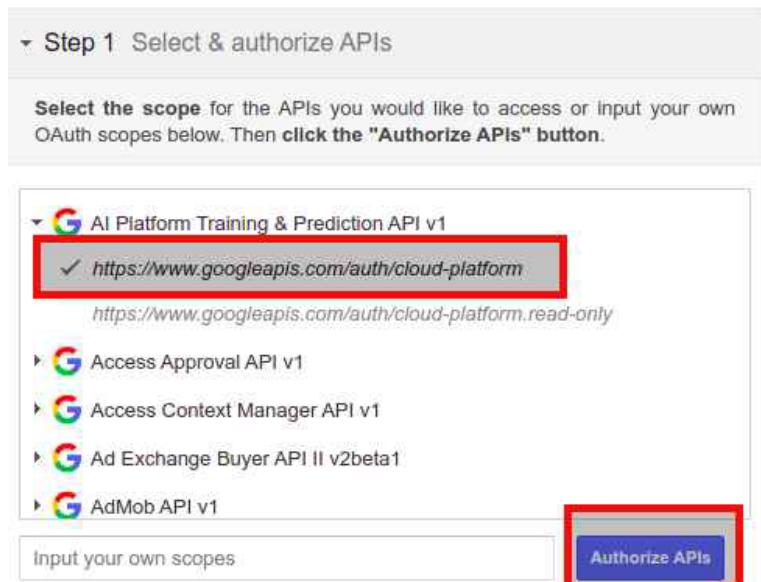
autenticación para la librería requests. En este otro [enlace](#) puedes consultar, de forma gráfica, los diferentes flujos de autenticación (en este caso con Google).

Se puede experimentar con el espacio OAuth 2 de Google. Por ejemplo, para el caso de AI Platform Training & Prediction API (para el entrenamiento de modelos de ML), existe un [servicio web](#) para explorar todos los endpoints y sus especificaciones (Discovery Document), bajo la siguiente URL:

[https://ml.googleapis.com/\\$discovery/rest?version=v1](https://ml.googleapis.com/$discovery/rest?version=v1)

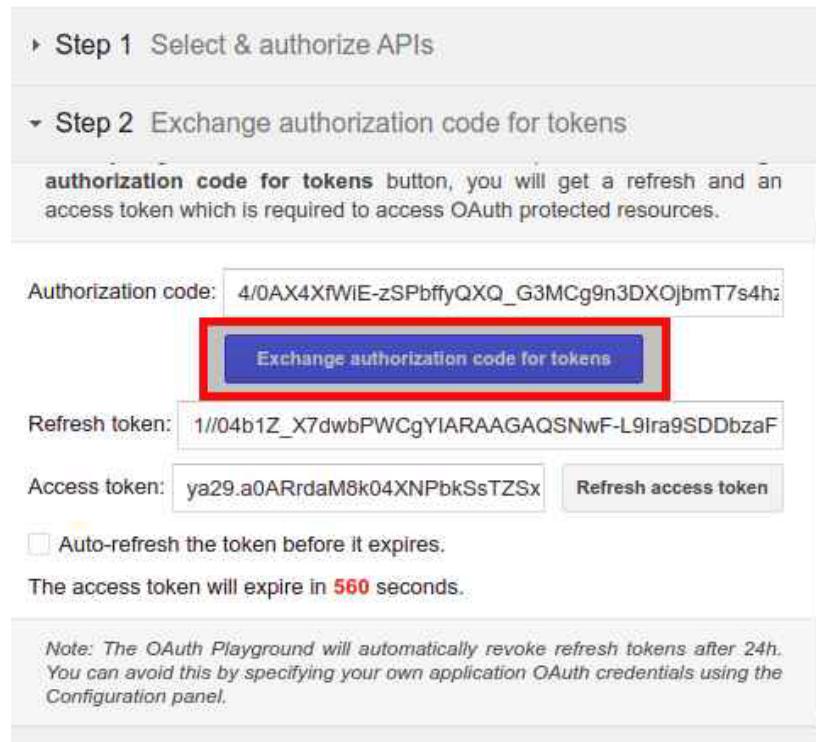
Aunque no es necesario expedir un token de acceso para explorar este servicio (se puede consultar con un simple navegador), vamos a realizar los pasos que se deberían dar para invocar cualquier otro servicio que necesitase un token. A continuación se describen los pasos realizados en el [espacio OAuth 2](#):

**PASO 1:** Seleccionamos la primera entrada del servicio "AI Platform Training & Prediction API v1" y pulsamos en "Authorize APIs" (bajo la sección "Select & authorize APIs"). Utiliza una cuenta de prueba que tengas de Google:



**PASO 2:** Tras pulsar "Authorize APIs" se despliega el siguiente paso (Exchange authorization code for tokens), donde ya viene informado el campo "Authorization code" (es el código que nos

envía en primera instancia la plataforma de Google), y a continuación ya podemos pulsar el botón "Exchange authorization code for tokens":



The screenshot shows the OAuth Playground interface at Step 2. It displays an authorization code (4/0AX4XfWIE-zSPbffyQXQ\_G3MCg9n3DXOjbmT7s4h2) and a refresh token (1//04b1Z\_X7dwbPWCgYIARAAGAQSFnF-L9lra9SDDbzaF). Below these, an access token (ya29.a0ARrdaM8k04XNPbkSsTZSx) is shown with a "Refresh access token" button. A checkbox for auto-refresh is present, and a note states the access token will expire in 560 seconds. A note at the bottom explains that refresh tokens are automatically revoked after 24 hours.

Authorization code: 4/0AX4XfWIE-zSPbffyQXQ\_G3MCg9n3DXOjbmT7s4h2

Exchange authorization code for tokens

Refresh token: 1//04b1Z\_X7dwbPWCgYIARAAGAQSFnF-L9lra9SDDbzaF

Access token: ya29.a0ARrdaM8k04XNPbkSsTZSx Refresh access token

Auto-refresh the token before it expires.  
The access token will expire in **560** seconds.

Note: The OAuth Playground will automatically revoke refresh tokens after 24h. You can avoid this by specifying your own application OAuth credentials using the Configuration panel.

Tras pulsar este botón, se informan los campos "Refresh token" y "Access token", y se indica el tiempo tras el cual caducará el Access token (en segundos). Si se pulsa en el botón "Refresh access token", se obtendrá un nuevo "Access token".

**PASO 3:** En este paso, ya estamos preparados para acceder al endpoint para el cual hemos solicitado el token de acceso. Por tanto, introducimos la URL en el campo "Request URI" (no necesitamos introducir el token de acceso en la cabecera según la nota que se indica abajo).

Pulsamos en "Send the request", y obtenemos la respuesta en la parte derecha del panel:

The screenshot shows the OAuth 2.0 Playground interface. On the left, there are three steps: Step 1 (Select & authorize APIs), Step 2 (Exchange authorization code for tokens), and Step 3 (Configure request to API). Step 3 is expanded, showing a form to construct an HTTP request. The 'Request URI' field contains [https://ml.googleapis.com/\\$discovery/rest?version=v1](https://ml.googleapis.com/$discovery/rest?version=v1). The 'Send the request' button is highlighted with a red box. To the right, under 'Request / Response', the response is shown in JSON format:

```

HTTP/1.1 200 OK
Content-length: 217963
X-xss-protection: 0
Content-location: https://ml.googleapis.com/$discovery/rest?version=v1
X-content-type-options: nosniff
Transfer-encoding: chunked
Vary: Origin, X-Origin, Referer
Server: ESF
Content-encoding: gzip
Cache-control: private
Date: Tue, 16 Nov 2021 15:36:32 GMT
X-frame-options: SAMEORIGIN
Alt-svc: h3=":443"; ma=2592000,h3-29=:443; ma=2592000,h3-0050=:443; ma=2592000,h3-0046=:443; ma=2592000,h3-Q043=:443; ma=2592000,quic=:443; ma=2592000; v="46,43"
Content-type: application/json; charset=UTF-8
  {
    "protocol": "rest",
    ...
  }
  
```

*Note: The OAuth access token in Step 2 will be added to the Authorization header of the request.*

En la parte derecha se puede apreciar que el token de acceso se ha enviado como parte del request, y que se ha obtenido un código 200 satisfactorio en la respuesta. El cuerpo del mensaje se puede ver en formato JSON:

**Request / Response**

Content-type: application/json; charset=UTF-8

```

{
  "protocol": "rest",
  "fullyEncodeReservedExpansion": true,
  "ownerName": "Google",
  "mtlsRootUrl": "https://ml.mtls.googleapis.com/",
  "batchPath": "batch",
  "id": "ml:v1",
  "description": "An API to enable creating and using machine learning models.",
  "ownerDomain": "google.com",
  "rootUrl": "https://ml.googleapis.com/",
  "parameters": {
    "upload_protocol": {
      "type": "string",
      "location": "query",
      "description": "Upload protocol for media (e.g. \\"raw\\", \\"multipart\\")."
    },
    "prettyPrint": {
      "default": "true",
      "type": "boolean",
      "description": "Returns response with indentations and line breaks.",
      "location": "query"
    },
    "access_token": {
      "type": "string",
      "description": "OAuth access token.",
      "location": "query"
    }
  }
}
  
```

### 3.7. Gestión de errores

En las llamadas a API REST es necesario implementar mecanismos para contemplar las situaciones en que se produzca algún tipo de imprevisto. Esto proporciona **robustez** a las aplicaciones que desarrollemos.

En primer lugar, es necesario conocer los **códigos de respuesta** que existen, y su significado.

Podrás encontrar una lista completa en este [enlace](#).

A continuación, tenemos varias posibilidades para capturar los posibles errores. Podemos realizar algún tipo de acción con un bloque condicional:

```
response = requests.get("http://api.open-notify.org/astros.json")
if (response.status_code == 200):
    print("La solicitud se ha realizado con éxito.")
    # Aquí insertar código cuando la solicitud es exitosa
elif (response.status_code == 404):
    print("No se ha encontrado ningún resultado.")
    # Aquí insertar código cuando la solicitud NO es exitosa
```

También podemos utilizar un bloque try/except:

```
try:
    response = requests.get('http://api.open-notify.org/astros.json')
    response.raise_for_status() # Lanzará una excepción si se ha producido un error que será capturado
    # por el bloque "except"
    # Código adicional, ejecutado si no se ha producido ningún error
except requests.exceptions.HTTPError as error:
    print(error)
    # Código a ejecutar si se ha producido un error
```

### 3.8. Descarga de HTML

Otra de las utilidades de la librería requests es poder obtener el código HTML de una página web.

Esto es especialmente importante para el caso de web scraping.

Para ello, bastará con realizar una petición de tipo GET a la URL cuyo código HTML queremos descargar, de la forma:

```
>>> rest = requests.get("http://webcode.me")
>>> print(rest.text)
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>My html page</title>
</head>
<body>

    <p>
        Today is a beautiful day. We go swimming and fishing.
    </p>

    <p>
        Hello there. How are you?
    </p>

</body>
</html>
```

## 4. DJANGO REST FRAMEWORK

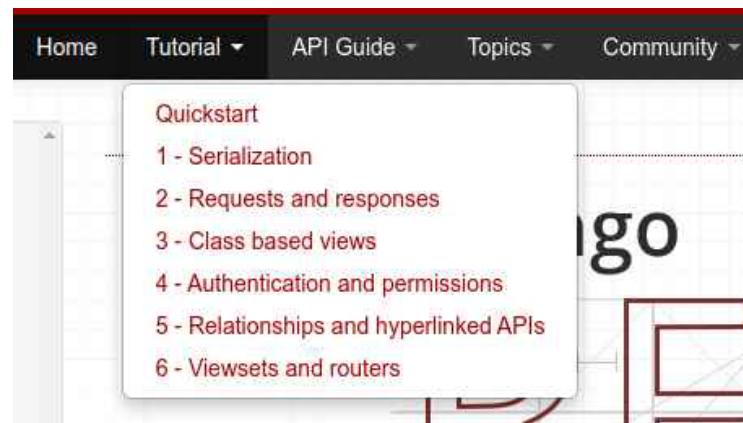
En los apartados anteriores hemos visto cómo consumir servicios REST mediante la librería `requests` de Python.

En esta segunda parte del documento vamos a aprender cómo crear servicios REST mediante el framework Django para poder desacoplar la capa backend de la capa cliente (frontend). Estos servicios REST podrán ser consumidos posteriormente por aplicaciones web cliente, apps móviles, aplicaciones de escritorio o incluso otros sistemas backend. Vamos a continuar utilizando, a modo ilustrativo, el ejemplo del portfolio.

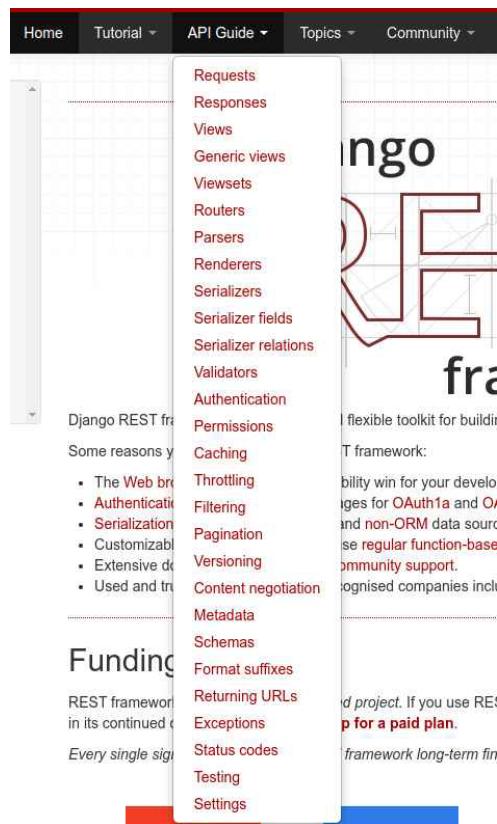
En los siguientes subapartados vamos a instalar la app de Django que nos permite programar servicios REST (Django REST Framework), desmenuzaremos las distintas partes en que se compone un servicio REST (serializadores, vistas y URLs), así como la herramienta de Django para poder probarlos de forma visual. Finalmente, estableceremos la forma de documentar la API REST para que los desarrolladores del lado cliente puedan utilizarla adecuadamente.

Toda la documentación referente a Django REST Framework la podrás encontrar en este [enlace](#).

Por una parte, tenemos un tutorial introductorio en el que podemos revisar lo más importante del framework, bajo el submenú Tutorial:



Y la documentación exhaustiva la encontramos bajo API Guide:



Existen también numerosos recursos en Internet, y videotutoriales que explican en profundidad la utilización de este framework.

#### 4.1. Instalación

Para poder utilizar Django REST Framework (Django REST Framework a partir de ahora), hemos de instalar la app correspondiente (junto con sus dependencias) mediante pip, según se detalla en este [enlace](#).

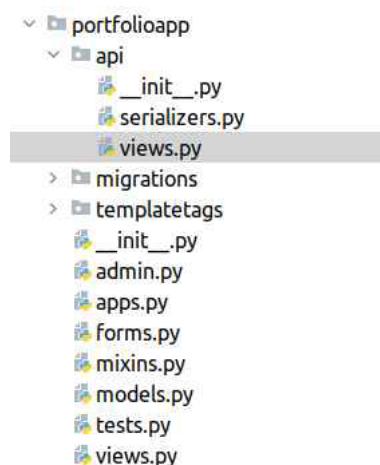
También se detalla en esta breve guía como configurar las URLs de DRF para probar los servicios REST mediante el API navegable.

Asegúrate de incluir lo siguiente en urls.py:

```
38
39     path('api/', include(router.urls)),
```

## 4.2. Organización de ficheros

Durante el desarrollo de los servicios web vamos a crear, como mínimo, dos tipos de objetos: serializadores y vistas. Para poder organizar estos nuevos objetos convenientemente y nuestras apps puedan ser exportables, vamos a crear un nuevo directorio "api" dentro de cada app para la que desarrollemos servicios REST. Sirva el siguiente como ejemplo:



Dentro de la nueva carpeta "api" crearemos un nuevo fichero `__init__.py`, y como mínimo habrá dos ficheros `serializers.py` y `views.py` (para los serializadores y vistas, correspondientemente).

## 4.3. Serializadores

Los serializadores son equivalentes, en el modelo vista-controlador, a los formularios de entrada de datos en el sistema. Funcionan en dos sentidos:

- Transforman tipos de datos complejos de Python, muchas veces extraídos de los modelos, en ficheros de texto plano en formato JSON o XML (entre otros). A este paso se le llama **serialización**.
- Conversión de datos JSON o XML a tipos complejos de datos en Python, previa validación

de los datos (como un formulario). A este paso se le llama **deserialización**.

En nuestro proyecto de portfolio, necesitamos serializadores para:

- Listado de proyectos
- Formulario de proyecto

Normalmente, utilizaremos un par serializador-vista para las listas, y otro par para las operaciones CRUD de un objeto. En el primer caso, consultaremos una lista (paginada o no) y necesitaremos menos campos; en el segundo caso, necesitaremos recuperar todo (o casi todos) los campos de un objeto, e implementar más validaciones para la consistencia de los datos.

Toda la información sobre serializadores la vas a encontrar en estos enlaces:

- [Serializers](#)
- [Serializer fields](#)
- [Serializer relations](#)

#### 4.3.1. Categorías

Dado que el modelo de categorías contiene un solo campo (el nombre) podemos crear un serializador que nos sirva tanto para listar categorías, como para acceder a su detalle. Vamos a empezar con este caso, que es el más sencillo.

Definimos un solo serializador de categorías del siguiente modo (en serializers.py):

```
1  from rest_framework import serializers
2  from portfolioapp.models import Proyecto, Categoria
3
4  class CategoriaSerializer(serializers.ModelSerializer):
5      class Meta:
6          model = Categoria
7          fields = (
8              'id',
9              'nombre'
10         )
```

Podríamos haber incluído solo el campo "nombre", pero es conveniente incluir el "id" por si lo necesitamos en el frontend para realizar cualquier tipo de operación con una categoría, para lo cual necesitaríamos utilizar su id concreto.

#### 4.3.2. Listado de proyectos

Una primera versión de este serializador sería la siguiente (insertamos el siguiente código en serializers.py de la app portfolioapp):

```
1  from rest_framework import serializers
2  from portfolioapp.models import Proyecto
3
4  class ProyectoListSerializer(serializers.ModelSerializer):
5      class Meta:
6          model = Proyecto
7          fields = [
8              'id',
9              'titulo',
10             'descripcion',
11             'fecha_creacion',
12             'imagen',
13             'categorias'
14         ]
```

Prácticamente estamos utilizando todos los campos del modelo Proyecto, con lo que vamos a introducir el siguiente cambio: el campo "descripcion" va a ser recortado para que incluya los 30 primeros caracteres (al ser una lista, necesitamos menos detalle). Para ello, hacemos las siguientes modificaciones:

```
1  from rest_framework import serializers
2  from portfolioapp.models import Proyecto
3
4  class ProyectoListSerializer(serializers.ModelSerializer):
5
6      descripcion_short = serializers.SerializerMethodField()
7
8      class Meta:
9          model = Proyecto
10         fields = (
11             'id',
12             'titulo',
13             #'descripcion',
14             'descripcion_short',
15             'fecha_creacion',
16             'imagen',
17             'categorias'
18         )
19
20     def get_descripcion_short(self, obj):
21         return obj.descripcion[:30]
```

Hemos realizado los siguientes cambios:

- Comentamos el campo 'descripcion' y añadimos el campo 'descripcion\_short'.
- Como 'descripcion\_short' no existe en el modelo, creamos un atributo adicional 'descripcion\_short' a partir de la clase SerializerMethodField, y lo incluimos en la lista 'fields'.
- Por último, creamos un método get\_descripcion\_short que recibe como parámetro el objeto que estamos tratando, y devuelve los 30 primeros caracteres de su descripción.

NOTA: Cuando definimos un SerializerMethodField, el método correspondiente que deriva su valor tendrá el nombre `get_[NOMBRE DEL CAMPO]`, y recibe como parámetro el objeto asociado al modelo definido en Meta.

En estos momentos aún no podemos probar la salida del serializador, pero si avanzamos en el tiempo y damos por supuesto que ya tenemos la vista correspondiente programada (y su URL), obtendríamos la siguiente salida:

```
[{"id": 1, "titulo": "Portfolio", "descripcion_short": "Aplicaci\u00f3n web para listar mis proyectos", "fecha_creacion": "2022-10-27T09:59:40+02:00", "imagen": "http://localhost/media/djang-views-types-1024x631.jpg", "categorias": [1, 4, 7, 10]}, {"id": 2, "titulo": "Avaluat", "descripcion_short": "Aplicaci\u00f3n web para la gesti\u00f3n", "fecha_creacion": "2022-12-01T16:24:00+01:00", "imagen": "http://localhost/media/RCZ4MPQSKFJ03060RPJBRYBEMKE.jpg", "categorias": [3, 5, 8]}, {"id": 23, "titulo": "nuevo", "descripcion_short": "nuevo", "fecha_creacion": "2022-11-21T13:08:43+01:00", "imagen": "http://localhost/media/Adri%C3%A1nVicenteMaci%C3%A1-Tarea-3.png", "categorias": [4]}, {"id": 24, "titulo": "nuevo", "descripcion_short": "nuevo", "fecha_creacion": "2022-11-19T14:01:03+01:00", "imagen": null, "categorias": [4]}, {"id": 25, "titulo": "asdf", "descripcion_short": "asdf", "fecha_creacion": "2022-11-21T12:31:10+01:00", "imagen": null, "categorias": [3]}]
```

Hemos hecho un gran avance. Hemos conseguido generar una lista de proyectos, con sus correspondientes campos, y uno de ellos es una versi\u00f3n recortada de la descripci\u00f3n del proyecto. Pero existe un aspecto que necesitamos cambiar: el campo "categorias" solo nos devuelve el ID de las categor\u00edas asociadas al proyecto. Lo ideal es que nos devolviese tanto el ID como su descripci\u00f3n, y as\u00ed podr\u00edamos visualizar (en el lado cliente) la lista con el nombre de las categor\u00edas. Para poder realizar esto, necesitamos serializar los objetos Categor\u00eda dentro de la lista de proyectos. Por tanto, se trata de un **serializador anidado**.

Para crear este serializador anidado, necesitamos utilizar el serializador de categorías que hemos definido anteriormente.

Con este serializador, vamos a poder serializar a nuestra conveniencia las categorías de cada proyecto, pero ¿cómo?. La respuesta es: definiendo un nuevo campo en ProyectoListSerializer de tipo SerializerMethodField (como description\_short) pero cuyo valor esté serializado a su vez por CategoriaSerializer. Modifica ProyectoListSerializer con el siguiente código:

```
12     class ProyectoListSerializer(serializers.ModelSerializer):
13
14         descripcion_short = serializers.SerializerMethodField()
15         categorias_serialized = serializers.SerializerMethodField()
16
17         class Meta:
18             model = Proyecto
19             fields = (
20                 'id',
21                 'titulo',
22                 #'descripcion',
23                 'descripcion_short',
24                 'fecha_creacion',
25                 'imagen',
26                 #'categorias',
27                 'categorias_serialized'
28             )
29
30         def get_descripcion_short(self, obj):
31             return obj.descripcion[:30]
32
33         def get_categorias_serialized(self, obj):
34             return CategoriaSerializer(obj.categorias, many=True, read_only=True).data
```

Ahora observamos la salida:

```

    "id": 1,
    "titulo": "Portfolio",
    "descripcion_short": "Aplicación web para listar mis",
    "fecha_creacion": "2022-10-27T09:59:40+02:00",
    "imagen": "http://localhost/media/djang-views-types-1024x631.jpg",
    "categorias_serialized": [
        {
            "id": 1,
            "nombre": "PHP"
        },
        {
            "id": 4,
            "nombre": "Django"
        },
        {
            "id": 7,
            "nombre": "React"
        },
        {
            "id": 10,
            "nombre": "Bootstrap"
        }
    ],
    "categorias": [
        {
            "id": 1,
            "nombre": "PHP"
        },
        {
            "id": 4,
            "nombre": "Django"
        },
        {
            "id": 7,
            "nombre": "React"
        },
        {
            "id": 10,
            "nombre": "Bootstrap"
        }
    ]
},
{
    "id": 2,
    "titulo": "Avaluar",
    "descripcion_short": "Aplicación web para la gestión",
    "fecha_creacion": "2022-12-01T16:24:00+01:00",
    "imagen": "http://localhost/media/RCZ4MPQSKFJ03O60RPJBREMKE.jpg",
    "categorias_serialized": [
        {
            "id": 3,
            "nombre": "Laravel"
        },
        {
            "id": 5,
            "nombre": "Flask"
        }
    ],
    "categorias": [
        {
            "id": 3,
            "nombre": "Laravel"
        },
        {
            "id": 5,
            "nombre": "Flask"
        }
    ]
},
{
    "id": 6,

```

Ahora sí vamos a poder utilizar este array JSON y utilizarlo desde código JavaScript para representar correctamente la lista de proyectos junto con sus categorías.

Entonces, ¿qué hemos hecho exactamente con el método `get_categorias_serialized`? Hemos utilizado el serializador `CategoríaSerializer` para transformar las categorías de un proyecto (que al final es una lista de los IDs de las categorías asociadas) a una lista de diccionarios JSON con el par `id/nombre`.

Esto nos lleva a la siguiente pregunta: ¿podríamos utilizar un serializador en cualquier parte de un código Python para serializar un determinado objeto? La respuesta es que sí, como se explica en este [videotutorial](#). Enfoca este vídeo estableciendo el paralelismo entre formularios de Django, y serializadores (`is_valid`, `validate` ...).

#### 4.3.3. Detalle de proyecto

Ahora queremos crear un serializador que nos sirva para poder modificar todos los campos de un

proyecto. Ya hemos desarrollado anteriormente el formulario de proyecto mediante el modelo vista-controlador:

 A screenshot of a web application interface. At the top, there's a navigation bar with links for 'Portfolio', 'INICIO', 'CATEGORÍAS ▾', and 'CONTACTO'. Below the navigation, the title 'Actualizar proyecto' is displayed. The form contains several input fields: 'Título\*' with the value 'Avaluat', 'Descripción\*' with the value 'Aplicación web para la gestión de la evaluación basada en criterios.', 'Fecha creación\*' with the value '01/12/2022 16:24:00', 'Año\*' with the value '2022', 'Categoria\*' with a dropdown menu showing options like '1 - PHP', '2 - Python', '3 - Laravel', and '4 - Django', and an 'Imagen' section with a file input field containing 'RCZ4MPQSKFJ03O6ORPJBYRBMKE.jpg' and a 'Choose File' button. At the bottom of the form are two buttons: a green 'Actualizar' button and a red 'Eliminar' button.

Por tanto hemos de incluir todos estos campos en el serializador:

```

37  class ProyectoDetailSerializer(serializers.ModelSerializer):
38
39      class Meta:
40          model = Proyecto
41          fields = (
42              'id',
43              'titulo',
44              'descripcion',
45              'fecha_creacion',
46              'year',
47              'imagen',
48              'categorias'
49          )
  
```

NOTA: aunque incluyamos el id de un objeto en el serializador, no vamos a poder modificarlo

desde el frontend. La columna "id" o "pk" siempre es de lectura.

#### 4.4. Vistas y Mixins

Las vistas en los servicios REST siguen cumpliendo el mismo objetivo que las vistas en el modelo vista-controlador: implementan la lógica de negocio y actúan sobre los datos introducidos mediante el serializador correspondiente (formularios, en el caso del modelo vista-controlador).

Cabe destacar también lo siguiente:

- Las vistas pueden o no tener asociado un serializador (de igual modo que las vistas del modelo vista-controlador, que no necesariamente han de tener un formulario asociado), dependiendo de su propósito. La misión de las vistas es recibir un request, y devolver un response, y muchas veces la lógica de negocio que implementan no requiere manejar ningún modelo para devolver información (se puede devolver una confirmación al finalizar un proceso).
- Al igual que en modelo vista-controlador, existen vistas basadas en función y vistas basadas en clase, con las mismas ventajas y desventajas.
- La tendencia es desarrollar los serializadores para que sean lo más simple posibles, y dejar el peso de la lógica de negocio a las vistas. Por ejemplo, se pueden establecer relaciones entre modelos en un serializador para actualizar diferentes modelos al mismo tiempo, pero las buenas prácticas aconsejan que sea la vista que recoge los datos, quien se encargue de hacer esto.

Primero vamos a revisar el aspecto de las vistas FBV, en este [enlace](#), para, a continuación, revisar el ejemplo más básico de una vista basada en la clase APIView, en este [enlace](#).

A continuación revisamos lo que son las vistas genéricas de DRF, en este [enlace](#), haciendo especial hincapié en los Mixins (ListModelMixin, CreateModelMixin, RetrieveModelMixin, UpdateModelMixin, DestroyModelMixin).

Por último, revisamos los [viewsets](#) (haciendo especial hincapié en el GenericViewSet).

Tras esta revisión, concluimos lo siguiente:

- Cuando hagamos operaciones CRUD sobre un modelo, utilizaremos la vista GenericViewSet en combinación con los mixins de modelo, según las operaciones que necesitemos hacer sobre dicho modelo.
- Para implementar lógicas de negocio que no sean particulares sobre un determinado modelo (por ejemplo, no involucren a ningún modelo o a varios de ellos), utilizaremos la vista APIView.

#### 4.4.1. Lista de categorías

Vamos a implementar nuestra primera vista, que va a ser la lista de categorías. El serializador ya lo tenemos, ahora falta elegir la vista más adecuada.

Podríamos pensar en utilizar una vista que herede de [ModelViewSet](#), ya que implementaría todas las operaciones CRUD sobre un determinado objeto (listado, creación, actualización, ...) y nos serviría para todo lo relacionado con dicho objeto. Pero en este caso estaríamos posibilitando realizar más operaciones que las realmente necesarias. Pensemos en el desplegable de categorías de la parte pública del portafolio: solo hemos de listar las categorías mediante un dropdown, no vamos a realizar ninguna operación más, por lo tanto no deberíamos exponer, mediante esta vista, más de lo necesario (creación, actualización, etc.).

Por tanto, necesitamos una vista que solo se dedique a mostrar el nombre de las categorías, nada más.

Dicho esto, vamos a utilizar la vista GenericViewSet en combinación con el mixin ListModelMixin para recuperar la lista de categorías. En api/views.py:

```

20   class CategoriaListViewSet(ListModelMixin,
21                             viewsets.GenericViewSet):
22     serializer_class = CategoriaSerializer
23     pagination_class = ShortResultsSetPagination
24
25     def get_queryset(self):
26       return Categoria.objects.all()

```

Ahora necesitamos configurar la URL correspondiente para poder comprobar el resultado. Para

ello, introducimos el siguiente código en urls.py:

```

21  from rest_framework import routers
22
23  from portfolioapp import views
24  from portfolioapp.api import views as api_views
25
26  router = routers.DefaultRouter()
27  router.register(r'categoria_list', api_views.CategoriaListViewSet, basename='categoria_list')

```

Más información sobre el objeto router, la encontrarás en este [enlace](#).

Para comprobar el resultado, consultamos la URL [http://localhost/api/categoria\\_list/](http://localhost/api/categoria_list/):

 A screenshot of a web browser displaying the Django REST framework's API browser. The title bar says "Django REST framework". The main content shows the "Categoria List List" endpoint at "GET /api/categoria\_list/". The response is an "HTTP 200 OK" with the following JSON data:
 

```

HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "nombre": "PHP"
    },
    {
        "id": 2,
        "nombre": "Python"
    },
    {
        "id": 3,
        "nombre": "Laravel"
    },
    {
        "id": 4,
        "nombre": "Django"
    },
    {
        "id": 5,
        "nombre": "Flask"
    },
    {
        "id": 6,
        "nombre": "Vue.js"
    },
    {
        "id": 7,
        "nombre": "React"
    },
    {
        "id": 8,
        "nombre": "Angular"
    },
    {
        "id": 9,
        "nombre": "Svelte"
    }
]
  
```

Enhorabuena, ¡has configurado tu primer servicio REST!

#### 4.4.2. Mantenimiento de categorías

En este apartado vamos a implementar el mantenimiento de categorías de dos formas: mediante la clase `ModelViewSet` y mediante la combinación de mixins con `GenericViewSet`.

Cabe hacer notar, en un primer lugar, que cualquier vista que implique alteración de los datos en la BBDD debería implementar mecanismos de seguridad para que solo los usuarios autorizados puedan realizar determinadas operaciones (excepto la de contacto, que es anónima).

Salvando el punto de la autenticación, que abordaremos en la siguiente unidad, vamos a implementar una vista CRUD sobre el objeto de categorías. Para ello, introducimos el siguiente código en `api/views.py`:

```

24     class CategoriaCRUDViewSet(viewsets.ModelViewSet):
25         serializer_class = CategoriaSerializer
26         queryset = Categoria.objects.all()
  
```

Y su correspondiente URL:

```

26     router = routers.DefaultRouter()
27     router.register(r'categoria_list', api_views.CategoriaListViewSet, basename='categoria_list')
28     router.register(r'categoria_crud', api_views.CategoriaCRUDViewSet, basename='categoria_crud')
  
```

Ahora, si consultamos la URL `http://localhost/api/categoria_crud/` veremos la lista de categorías, y al final de la página veremos un pequeño formulario para introducir nuevas categorías:

The screenshot shows a browser window with the URL `localhost/api/categoria_crud/`. The page title is "Django REST framework". On the left, there is a JSON representation of a list of categories:

```

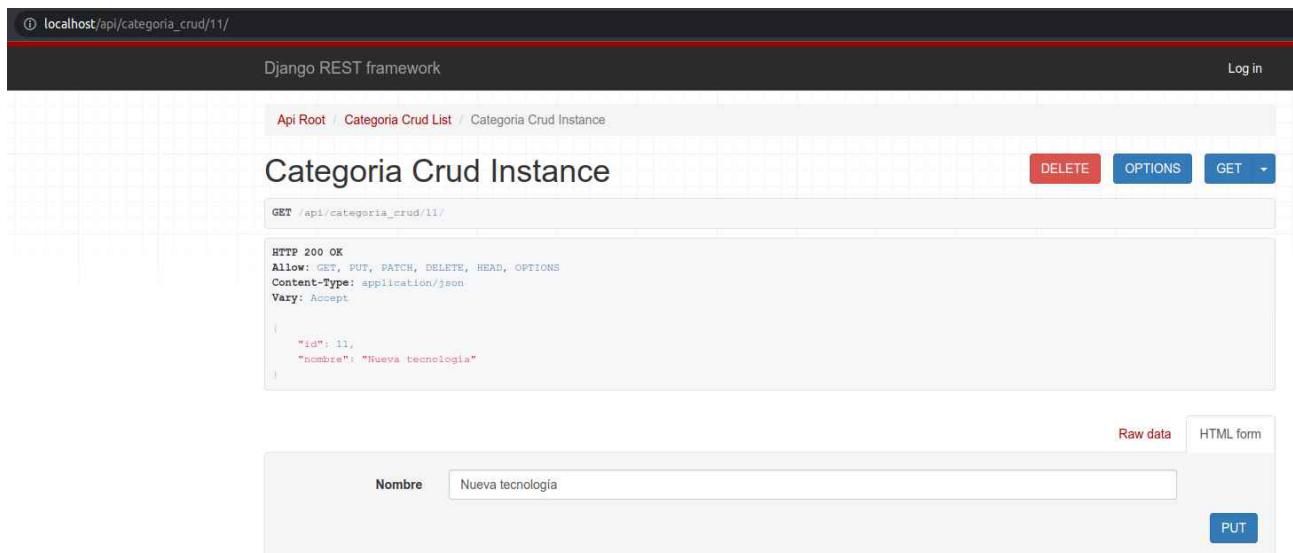
[{"id": 1, "nombre": "Python"}, {"id": 2, "nombre": "PHP"}, {"id": 3, "nombre": "Laravel"}, {"id": 4, "nombre": "Django"}, {"id": 5, "nombre": "Flask"}, {"id": 6, "nombre": "Vue.js"}, {"id": 7, "nombre": "React"}, {"id": 8, "nombre": "Angular"}, {"id": 9, "nombre": "Svelte"}, {"id": 10, "nombre": "Bootstrap"}]
  
```

On the right, there is a form with a single input field labeled "Nombre" and a "POST" button. Below the form, there are two tabs: "Raw data" and "HTML form".

En la pestaña "Raw data" podríamos introducir los datos en formato JSON para crear una nueva categoría, tras pulsar el botón POST. Esto nos evita utilizar herramientas adicionales, como

## Postman.

Creamos una nueva tecnología en el aula, y volvemos a visualizar la lista entera de categorías. Comprobamos que se ha creado correctamente. Pero, ¿cómo podríamos actualizarla o eliminarla? Para ello tomamos el ID de la nueva categoría, y anexamos dicho ID a la URL anterior, de la forma:



localhost/api/categoria\_crud/11/

Django REST framework

Log in

Api Root / Categoria Crud List / Categoria Crud Instance

DELETE OPTIONS GET

Categoria Crud Instance

GET /api/categoria\_crud/11/

HTTP 200 OK

Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS

Content-Type: application/json

Vary: Accept

{  
 "id": 11,  
 "nombre": "Nueva tecnologia"  
}

Raw data HTML form

Nombre: Nueva tecnología

PUT

Vemos que en la URL indica el ID 11 (es solo un ejemplo de un ID existente), y además ahora en el formulario el botón dice PUT (modificación) en lugar de POST (creación), y se habilita el botón DELETE.

Por tanto, hemos creado un servicio REST que implementa todas las operaciones CRUD sobre el objeto Categoría.

¿Cómo podríamos restringir determinadas operaciones CRUD sobre este modelo? Por ejemplo, queremos poder crear, recuperar y actualizar categorías, pero no borrarlas. Para ello, utilizaríamos la vista GenericViewSet en combinación con los mixins adecuados, como se muestra en el siguiente código (insértalo en api/views.py):

```

class CategoriaCreateRetrieveUpdateViewSet(mixins.CreateModelMixin,
                                           mixins.RetrieveModelMixin,
                                           mixins.UpdateModelMixin,
                                           viewsets.GenericViewSet):
    serializer_class = CategoriaSerializer

    def get_queryset(self):
        return Categoria.objects.all()
  
```

Creamos la correspondiente URL:

```

26 router = routers.DefaultRouter()
27 router.register(r'categoria_list', api_views.CategoriaListViewSet, basename='categoria_list')
28 router.register(r'categoria_crud', api_views.CategoriaCRUDViewSet, basename='categoria_crud')
29 router.register(r'categoria_create_retrieve_update', api_views.CategoriaCreateRetrieveUpdateViewSet, basename='categoria_create_retrieve_update')
  
```

Ahora, accedemos al detalle de una categoría y comprobamos que no aparece el botón DELETE, y vemos las posibles operaciones según el directiva **Allow**:

 A screenshot of the Django REST framework's browsable API interface. The URL is 'localhost/api/categoria\_create\_retrieve\_update/1'. The page title is 'Categoria Create Retrieve Update Instance'. Below the title, there is a 'GET /api/categoria\_create\_retrieve\_update/1/' button. The response section shows the following JSON data:
 

```

HTTP 200 OK
Allow: GET, PUT, PATCH, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "id": 1,
    "nombre": "PHP"
}
  
```

 At the bottom, there is a form with a 'Nombre' input field containing 'PHP' and a 'PUT' button. There are also 'Raw data' and 'HTML form' buttons above the form.

#### 4.4.3. Lista de proyectos

Como actividad de clase, implementamos la vista que liste todos los proyectos, y que utilice el serializador ProyectoListSerializer.

#### 4.4.4. Mantenimiento de proyectos

Para el servicio web de mantenimiento de proyectos (consulta, creación, actualización, borrado) vamos a utilizar el serializador ProyectoDetailSerializer que ya hemos programado anteriormente.

Insertamos la siguiente vista en api/views.py:

```

51     class ProyectoCRUDViewSet(mixins.CreateModelMixin,
52                               mixins.RetrieveModelMixin,
53                               mixins.UpdateModelMixin,
54                               mixins.DestroyModelMixin,
55                               viewsets.GenericViewSet):
56         serializer_class = ProyectoDetailSerializer
57         queryset = Proyecto.objects.all()

```

Configuramos la URL:

```

31     router.register(r'proyecto_list', api_views.ProyectoListViewSet, basename='proyecto_list')
32     router.register(r'proyecto_crud', api_views.ProyectoCRUDViewSet, basename='proyecto_crud')
33

```

Y probamos con un ID de proyecto existente:

 A screenshot of a web browser displaying a JSON response from a Django REST framework endpoint. The URL is 'localhost/api/proyecto\_crud/1/'. The response shows a single project instance with the following data:
 

```

{
    "id": 1,
    "titulo": "Portfolio",
    "descripcion": "Aplicación web para listar mis proyectos",
    "fecha_creacion": "2022-10-27T09:59:40+02:00",
    "year": 2022,
    "imagen": "http://localhost/media/djang-views-types-1024x631.jpg",
    "categorias": [
        1,
        4,
        7,
        10
    ]
}
  
```

 The browser interface includes a header bar with 'localhost/api/proyecto\_crud/1/' and 'Django REST framework'. Below the header is a navigation bar with 'Log in'. The main content area has a title 'Proyecto Crud Instance' and three buttons: 'DELETE' (red), 'OPTIONS' (blue), and 'GET' (blue). The bottom of the page shows the raw JSON response with its status code, headers, and content.

Si vamos al final de la página veremos un formulario para poder realizar las operaciones CRUD con el proyecto y probar el servicio web:

C localhost/api/proyecto\_crud/1/

Django REST framework Log in

Raw data HTML form

Título	Portfolio
Descripción	Aplicación web para listar mis proyectos
Fecha creación	mm/dd/yyyy, --:-- --
Año	2022
Imagen	<input type="file"/> Choose File No file chosen
Categoría	1 - PHP 2 - Python 3 - Laravel <b>4 - Django</b> 5 - Flask 6 - Vue.js 7 - React 8 - Angular 9 - Gatsby
<input type="button" value="PUT"/>	

También podemos modificar directamente el JSON mediante la pestaña Raw data:

C localhost/api/proyecto\_crud/1/

Django REST framework Log in

Raw data HTML form

Media type:	<input type="text" value="application/json"/>
Content:	<pre>{   "id": 1,   "titulo": "Portfolio",   "descripcion": "Aplicación web para listar mis proyectos",   "fecha_creacion": "2022-10-27T09:59:40+02:00",   "year": 2022,   "imagen": "http://localhost/media/djang-views-types-1024x631.jpg",   "categorias": [     1,     4,     7,     10   ] }</pre>
<input type="button" value="PUT"/> <input type="button" value="PATCH"/>	

#### 4.4.5. Métodos de las vistas

Tal y como vimos en las CBV del modelo vista-controlador, existen determinados métodos que podemos sobreescribir para modificar el comportamiento de las vistas estándar.

En este [enlace](#) se citan los posibles métodos, que revisamos en la clase. Iremos utilizando algunos de estos métodos en los siguientes apartados.

#### 4.4.6. Validaciones

En este apartado vamos a ver cómo podríamos, por ejemplo, comprobar que la fecha de creación pertenece al mismo año que el enviado en el campo "year". Esta comprobación la realizaremos en la creación y en la actualización del proyecto.

Para ello, vamos a sobreescibir los métodos `perform_create` y `perform_update` de la vista `ProyectoCRUDViewSet`, del siguiente modo:

```

69     def perform_create(self, serializer):
70         year = serializer.validated_data.get('year')
71         fecha_creacion = serializer.validated_data.get('fecha_creacion')
72         if year != fecha_creacion.year:
73             raise ValidationError(
74                 {'non_field_errors': "El año ha de coincidir con el de la fecha de creación"})
75         )
76         serializer.save()
77
78     def perform_update(self, serializer):
79         year = serializer.validated_data.get('year')
80         fecha_creacion = serializer.validated_data.get('fecha_creacion')
81         if year != fecha_creacion.year:
82             raise ValidationError(
83                 {'non_field_errors': "El año ha de coincidir con el de la fecha de creación"})
84         )
85         serializer.save()

```

Hemos de hacer la importación previamente de `ValidationError`:

```

10     from rest_framework.exceptions import ValidationError
11

```

NOTA: Como se ve, podemos crear un método (lo podemos llamar según nuestra conveniencia) que reproveche mucho código que se repite en los dos métodos, e invocarlo desde `perform_create` y `perform_update`. Lo realizamos como ejercicio de clase.

Vamos a probar que salta el error. Introducimos un año diferente al de la fecha de creación:

Django REST framework Log in

Raw data  HTML form

Título	Portfolio
Descripción	Aplicación web para listar mis proyectos
Fecha creación	01/18/2023, 05:46 PM <span style="float: right;">x</span>
Año	2022
Imagen	<input type="button" value="Choose File"/> No file chosen
Categoría	1 - PHP 2 - Python 3 - Laravel <b>4 - Django</b> 5 - Flask 6 - Vue.js 7 - React 8 - Angular ... <span style="float: right;">▼</span>
<input type="button" value="PUT"/>	

Al pulsar en el botón PUT, obtenemos el siguiente error:

Django REST framework Log in

[Api Root](#) / [Proyecto Crud List](#) / [Proyecto Crud Instance](#)

### Proyecto Crud Instance

[DELETE](#) [OPTIONS](#) [GET](#) ▾

[PUT /api/proyecto\\_crud/1/](#)

```
HTTP 400 Bad Request
Allow: GET, PUT, PATCH, DELETE, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "non_field_errors": "El año ha de coincidir con el de la fecha de creación"
}
```

Ésta será la respuesta HTTP que se devolverá a la parte front, con código 400, y con un JSON en el body que contenga el diccionario con los errores encontrados. Estos errores tendrán que ser procesados por la parte cliente y representados convenientemente.

NOTA: también disponemos del método `perform_destroy`, para el caso del mixin `DestroyModelMixin`, si fuese necesario.

#### 4.4.7. Parámetros

Imaginemos que pretendemos filtrar los proyectos posteriores a un determinado año. Esto lo podríamos realizar enviando un parámetro a la vista `ProyectoListViewSet` que contenga el año. Este parámetro será recogido en el método `get_queryset`, el cual devolverá un queryset filtrando

por el número de año. Veamos cómo queda la lógica de este método:

```
class ProyectoViewSet(mixins.ListModelMixin,
                      viewsets.GenericViewSet):
    serializer_class = ProyectoSerializer

    def get_queryset(self):
        year_from = self.request.query_params.get('year_from')
        if year_from:
            return Proyecto.objects.filter(year_dte=year_from)
        return Proyecto.objects.all()
```

Y el resultado, con el parámetro en la URL:

 A screenshot of the Django REST framework's API browser. The top navigation bar says "Django REST framework" and "Api Root / Proyecto List List". Below that, the main title is "Proyecto List List". A button labeled "GET /api/proyecto\_list/?year\_from=2021" is shown. The response section starts with "HTTP 200 OK" and includes headers: "Allow: GET, HEAD, OPTIONS", "Content-Type: application/json", and "Vary: Accept". The response body shows a JSON array with one item, representing a project object:
 

```
{
    "id": 1,
    "titulo": "Portfolio",
    "descripcion_short": "Aplicación web para listar mis",
    "fecha_creacion": "2022-10-27T09:59:40+02:00",
    "imagen": "http://localhost/media/Screenshot_from_2023-01-18_16-47-51.png",
    "categorias_serialized": []
}
```

Como actividad en clase, vamos a realizar el filtrado por una determinada categoría.

#### 4.4.8. Ordenación

Para establecer la ordenación de una lista de objetos podríamos utilizar parámetros para enviar por URL el tipo de ordenación que necesitamos. Pero DRF nos proporciona un backend de ordenación que nos permite realizar esto con la mínima lógica.

Para establecer la ordenación **por defecto** de un queryset por el campo "fecha\_creacion", basta con establecer los atributos filter\_backends y ordering en la vista correspondiente, de la forma:

```
class ProyectoViewSet(mixins.ListModelMixin,
                      viewsets.GenericViewSet):
    serializer_class = ProyectoListSerializer
    filter_backends = (filters.OrderingFilter, )
    ordering = 'fecha_creacion'

    def get_queryset(self):
        year_from = self.request.query_params.get('year_from')
        if year_from:
            return Proyecto.objects.filter(year__gte=year_from)
        return Proyecto.objects.all()
```

De esta forma, consultamos el resultado de este endpoint y podemos comprobar que se ha ordenado por fecha de creación.

Si quisiésemos poder ordenar por varios campos, de forma ascendente o descendente, hemos de utilizar el atributo "ordering\_fields" de la forma:

```
class ProyectoViewSet(mixins.ListModelMixin,
                      viewsets.GenericViewSet):
    serializer_class = ProyectoListSerializer
    filter_backends = (filters.OrderingFilter, )
    ordering = 'fecha_creacion'
    ordering_fields = ['fecha_creacion', 'titulo']

    def get_queryset(self):
        year_from = self.request.query_params.get('year_from')
        if year_from:
            return Proyecto.objects.filter(year__gte=year_from)
        return Proyecto.objects.all()
```

Ahora, al consultar el endpoint con el API navegable de DRF, aparecerá un nuevo botón llamado "Filtros", que nos permitirá establecer la ordenación que pretendamos:

Django REST framework

Api Root / Proyecto List List

## Proyecto List List

GET /api/proyecto\_list/

HTTP 200 OK  
Allow: GET, HEAD, OPTIONS  
Content-Type: application/json  
Vary: Accept

```

{
    "id": 1,
    "titulo": "Portfolio",
    "descripcion_short": "Aplicación web para listar mis",
    "fecha_creacion": "2022-10-27T09:59:40+02:00"
}
  
```

**Filters**

**Ordenamiento**

- fecha\_creacion - ascendiente
- fecha\_creacion - descendiente
- titulo - ascendiente
- titulo - descendiente

Filtros

Al pulsar sobre alguna de las opciones, veremos que la URL se modifica convenientemente para acomodar un nuevo parámetro o parámetros de ordenación:

localhost/api/proyecto\_list/?ordering=-titulo

Django REST framework

Api Root / Proyecto List List

## Proyecto List List

GET /api/proyecto\_list/?ordering=-titulo

HTTP 200 OK

#### 4.4.9. Búsqueda

Al igual que en el caso de la ordenación, podríamos utilizar parámetros en el método `get_queryset` para poder establecer una búsqueda por determinados campos. Pero DRF también nos proporciona un mecanismo para poder buscar por campos con el mínimo código, utilizando el `SearchFilter` backend. Para ello, modificamos de nuevo la vista y agregamos el `SearchFilter` backend a la tupla de `filter_backends`, y definimos el atributo `search_fields`, de la forma:

```

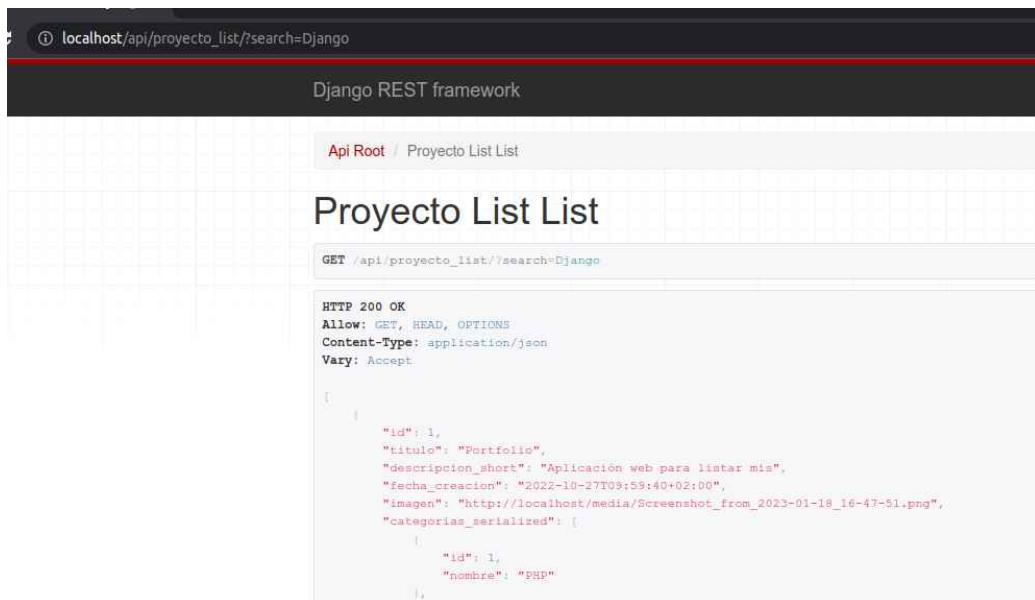
class ProyectoListViewSet(mixins.ListModelMixin,
                           viewsets.GenericViewSet):
    serializer_class = ProyectoListSerializer
    filter_backends = (filters.OrderingFilter, filters.SearchFilter)
    ordering = ['fecha_creacion']
    ordering_fields = ['fecha_creacion', 'titulo']
    search_fields = ['titulo', 'descripcion', 'categorias_nombre']

    def get_queryset(self):
        year_from = self.request.query_params.get('year_from')
        if year_from:
            return Proyecto.objects.filter(year__gte=year_from)
        return Proyecto.objects.all()
  
```

En este caso estamos permitiendo la búsqueda por título y descripción, y además por el nombre de una categoría asociada al proyecto. En el API navegable, vemos a que al pulsar sobre el botón Filtros nos aparece un nuevo campo Search. Al introducir un valor en este campo, se añade un nuevo parámetro "search" a la URL:

The screenshot shows the Django REST framework's API browser interface. On the left, the URL is `Api Root / Proyecto List List`. The main area displays the JSON response for a GET request to `/api/proyecto_list/`, which includes fields like `id`, `titulo`, `descripcion_short`, `fecha_creacion`, `imagenes`, and `categorias_serialized`. To the right, a modal window titled "Filters" is open, showing two sections: "Ordenamiento" (Sorting) and "Buscar" (Search). The "Ordenamiento" section has a dropdown menu with four options: "fecha\_creacion - ascendiente" (selected), "fecha\_creacion - descendiente", "titulo - ascendiente", and "titulo - descendiente". The "Buscar" section contains a search input field and a "Buscar" button.

Y la URL:



#### 4.4.10. Paginación

Otra de las funcionalidades que hemos ido desarrollando a lo largo de los proyectos del curso ha sido la paginación de las listas. En este aspecto, DRF también nos proporciona un método sencillo para paginar nuestras listas, basándonos en este [enlace](#).

Como se explica en la documentación, existen varios métodos para establecer el tamaño de la paginación. Nos interesa tener varios tamaños de paginación para poder utilizarlos según nos sea conveniente. Para ello, creamos un nuevo fichero `pagination.py` en un nuevo directorio "api" dentro de la app "common" con tres tipos de paginación:

- `LargeResultsSetPagination`: tamaño 50 por página
- `StandardResultsSetPagination`: tamaño 20 por página
- `ShortResultsSetPagination`: tamaño 5 por página

Creamos las clases correspondientes como actividad de clase, dentro de la app `portfolioapp`.

```
1  from rest_framework.pagination import PageNumberPagination
2
3
4  class LargeResultsSetPagination(PageNumberPagination):
5      page_size = 50
6      page_size_query_param = 'page_size'
7      max_page_size = 50
8
9
10     class StandardResultsSetPagination(PageNumberPagination):
11         page_size = 20
12         page_size_query_param = 'page_size'
13         max_page_size = 20
14
15
16     class ShortResultsSetPagination(PageNumberPagination):
17         page_size = 5
18         page_size_query_param = 'page_size'
19         max_page_size = 5
```

Ahora utilizamos una de estas clases dentro de la vista CategoriaListViewSet:

Y comprobamos la salida paginada en el API navegable:

```
17  from portfolioapp.api.pagination import LargeResultsSetPagination, StandardResultsSetPagination, ShortResultsSetPagination
18
19  class CategoriaListViewSet(mixins.ListModelMixin,
20                             viewsets.GenericViewSet):
21      model = Categoria
22      serializer_class = CategoriaSerializer
23      pagination_class = ShortResultsSetPagination
24
25      def get_queryset(self):
26          return Categoria.objects.all()
```

Django REST framework Log in

[Api Root](#) / Categoria List List

## Categoria List List

[OPTIONS](#) [GET](#) ▾

« [1](#) [2](#) [3](#) »

```
GET /api/categoria_list/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "count": 11,
    "next": "http://localhost/api/categoria_list/?page=2",
    "previous": null,
    "results": [
        {
            "id": 1,
            "nombre": "PHP"
        },
        {
            "id": 2,
            "nombre": "Python"
        },
        {
            "id": 3,
            "nombre": "Laravel"
        },
        {
            "id": 4,
            "nombre": "Django"
        },
        {
            "id": 5,
            "nombre": "Flask"
        }
    ]
}
```

#### 4.4.11. Vistas sin serializador

Es posible que necesitemos programar una vista que solo sea el punto de entrada para ejecutar una parte de nuestra lógica de negocio, sin necesidad de recibir o devolver datos. Por ejemplo: un proceso en nuestra aplicación `avaluapp` que se encargue de calcular las notas finales del alumnado a partir de las notas parciales de los criterios de evaluación, en cada una de las unidades.

En este caso, no necesitamos recibir ningún dato ni devolverlo, simplemente hemos de recibir la petición de un usuario autorizado, y devolver una respuesta de confirmación cuando se haya ejecutado correctamente.

En este caso no necesitamos serializar ni deserializar estructuras de datos (las operaciones se harían directamente en la base de datos). Por tanto, éste sería un claro ejemplo de una vista que implementa una lógica de negocio, pero no utiliza los datos recibidos para ejecutarla.

En este apartado vamos a realizar un ejemplo básico de vista sin serializador, con el fin de convertir todos los nombres de las categorías para que la primera letra sea mayúscula y las demás minúsculas. Esta operación la podríamos realizar de otras formas más efectivas, pero sirva como ejemplo ilustrativo.

Vamos a realizar la misma vista mediante CBV y FBV, con el método GET y sin parámetros.

Para el caso de CBV utilizamos la vista [APIView](#), que es la más sencilla de las CBV de DRF. La vista sería del siguiente modo:

```
86     class CapitalizeCategoriaView(APIView):
87
88         def get(self, request, format=None):
89             categorias = Categoria.objects.all()
90             for categoria in categorias:
91                 categoria.nombre = categoria.nombre.capitalize()
92                 categoria.save()
93             return Response(status=status.HTTP_200_OK)
```

Ahora creamos la URL, pero no la podemos registrar con el router como las demás vistas basadas en vistas genéricas, hemos de configurarla en la lista urlpatterns, como las configuradas en el modelo vista-controlador, de la forma:

```
39
40     path(r'api/categoría_capitalize', api_views.CapitalizeCategoriaView.as_view(), name='categoría_capitalize'),
```

Ahora vamos a programar una vista basada en función que realice lo mismo. Previamente hemos de importar el decorador `api_view`:

```
from rest_framework.decorators import api_view
```

Y programamos la vista de la siguiente forma:

```
97     @api_view(['GET'])
98     def capitalize_categoria_view(request):
99         if request.method == "GET":
100             categorias = Categoria.objects.all()
101             for categoria in categorias:
102                 categoria.nombre = categoria.nombre.capitalize()
103                 categoria.save()
104             return response.Response(status=status.HTTP_200_OK)
```

Por último, configuramos la URL en la lista urlpatterns:

```
41
42     path(r'api/categoría_capitalize_fbv', api_views.capitalize_categoria_view, name='categoría_capitalize_fbv'),
43
```

#### 4.5. API navegable de DRF: test y documentación

DRF nos proporciona de serie una herramienta para poder probar y documentar los servicios web programados, llamada "Browsable API", que hemos ido utilizando a lo largo de este tutorial para verificar los resultados.

En este [enlace](#) encontrarás todo lo referente a la documentación de servicios REST en DRF, tanto lo que concierne a módulos externos con los que poder documentar nuestras APIs como el API navegable.