

Aplicación web CRUD con Django, JavaScript y Ajax para crear, leer, actualizar y borrar datos de forma asíncrona.

Autor: Marco Antonio Irala Ortellado

Resumen

En este trabajo se enfoca más bien en ser una guía para aquellas personas que se encuentran estudiando el framework Django y el lenguaje de programación JavaScript para crear una aplicación web que realice sus operaciones fundamentales de Crear, editar, actualizar y eliminar datos de una base de datos, la cual se utiliza generalmente el acrónimo CRUD, a través de peticiones y solicitudes POST con Ajax. Tal abordaje es necesario a fin de que pueda facilitar a aquellas personas a tener una guía que les sea útil de alguna manera y le pueda ayudar en sus metas. El objetivo principal de este trabajo es el de crear una aplicación web CRUD con Django y JavaScript realizando las comunicaciones entre el front-end y el back-end a través de peticiones post con Ajax. La metodología aplicada es la de ir realizando la aplicación web a medida que desarrollamos el trabajo, con explicaciones sencillas y demostraciones, siempre utilizando y proporcionando las fuentes de las documentales de cada herramienta que vayamos a utilizar. Este trabajo demostrará cómo realizar una aplicación web CRUD con Django y JavaScript realizando las comunicaciones entre el front-end y el back-end a través de peticiones post con Ajax.

Palabras clave: Django, JavaScript, Ajax, Peticiones.

Introducción

Este trabajo tiene como objetivo principal el crear una aplicación web que realice sus operaciones fundamentales de Crear, editar, actualizar y eliminar datos de una base de datos, la cual se utiliza generalmente el acrónimo CRUD, a través de peticiones y solicitudes POST con Ajax.

Tal abordaje es necesario a fin de que pueda facilitar a aquellas personas a tener una guía que les sea útil de alguna manera y le pueda ayudar en sus metas.

La metodología aplicada es la de ir realizando la aplicación web a medida que desarrollamos el trabajo, con explicaciones sencillas y demostraciones, siempre utilizando y proporcionando las fuentes de las documentales de cada herramienta que vayamos a utilizar.

Primeramente descargamos los recursos necesarios para la creación de nuestra aplicación web, para ello necesitamos la última versión de Django que lo instalaremos a través de la línea de comandos o desde nuestro editor de código preferencial los cuales pueden ser Visual Studio Code, Vim, Sublime Text etc.

Luego crearemos un proyecto nuevo con Django, con los comandos necesarios, asimismo dentro del proyecto crearemos una aplicación que puede ser un blog, un administrador de tareas o un registro de boletos de autobús.

Creado nuestra aplicación, realizaremos las configuraciones necesarias para incorporarlo dentro de nuestro proyecto y poder ejecutarlo, hecho esto crearemos los modelos necesarios para nuestra base de datos, el cual solo consta de un solo campo por razones prácticas.

Dentro de nuestra aplicación crearemos carpetas necesarias para nuestro proyecto los cuales son el templates y static los que contendrán nuestros archivos html y js para la interfaz y funcionamiento de nuestra aplicación

Crearemos las vistas el cual es una función de python que toma solicitudes web y devuelve una respuesta web, prácticamente puede devolver cualquier cosa (XML, imagen, error 404 etc) .

Luego crearemos un archivo html con una estructura básica, dentro del cual crearemos una tabla que contendrá algunas columnas para mostrar nuestros datos de la base de datos obtenida a través de solicitudes POST con Ajax y realizar ciertas acciones, para ello utilizaremos una herramienta muy útil llamada datatable, consistente en una extensión de jQuery que nos permite pintar tablas con paginado, búsqueda, ordenar por columnas, etc.

Crearemos dentro de nuestro archivo html dos botones, uno crear categoría nueva y el otro actualizar nuestra tabla asincrónicamente, es decir sin necesidad de cargar la página completa, únicamente los datos que se muestran. Asimismo al momento de crear un nuevo

registro dentro de nuestra vista, lo valoraremos y procesaremos con los datos enviados por el método post, para verificar y realizar la acción correspondiente. -

Posteriormente, una vez que podamos registrar una nueva categoría, vamos a editarlo, seleccionando los datos que deseamos modificar presionando un botón, el cual lo situamos dentro de nuestra tabla en cada fila dentro de la columna opciones.

Finalmente, al tener la posibilidad de crear, editar y leer datos de nuestra base de datos, nos queda la opción de eliminar un registro para completar nuestra aplicación CRUD, esto lo hacemos seleccionando los datos que deseamos eliminar presionando un botón, el cual lo situamos dentro de nuestra tabla en cada fila dentro de la columna opciones.

Metodología

La metodología utilizada en esta guía teórico-práctico, es decir, iremos creando nuestra aplicación y explicando sus distintas partes de una forma sintetizada, proporcionando por el camino los recursos bibliográficos o fuentes de donde se extrajo los datos y las documentales de las distintas herramientas que vayamos a utilizar en esta guía, esto con el fin de crear una aplicación web que realice sus operaciones fundamentales de Crear, editar, actualizar y eliminar datos de una base de datos, la cual se utiliza generalmente el acrónimo CRUD, a través de peticiones y solicitudes POST con Ajax.

UN PROYECTO NUEVO CON DJANGO

Creando un nuevo proyecto con Django: para esto necesitamos descargarlo desde el sitio oficial de django <https://www.djangoproject.com/download/>, para ello les recomiendo siempre leer los documentos oficiales, django está muy bien documentado y cuenta con ejemplos y mucha guía para su correcto uso.

Para este proyecto usaremos Visual Studio Code para nuestro editor de código fuente, una vez instalado nuestra última versión de Django, vayamos a nuestro editor y en una terminal nueva, creamos una carpeta donde estará ubicada nuestro proyecto, en mi caso le he puesto el nombre de proyecto-django, dentro del mismo creamos nuestro proyecto.

Cabe aclarar que doy por sentado que el lector de esta guía tiene conocimientos básicos de instalación y configuración de django a fin de no hacerlo tedioso y extenso. -

```
# django-admin startproject config .  
  
#bash-3.2$ django-admin startproject config .  
#bash-3.2$ ls  
config          manage.py
```

Una vez creado el proyecto nuevo, creamos una aplicación dentro del mismo, que será el encargado de gestionar las funcionalidades de nuestro proyecto.

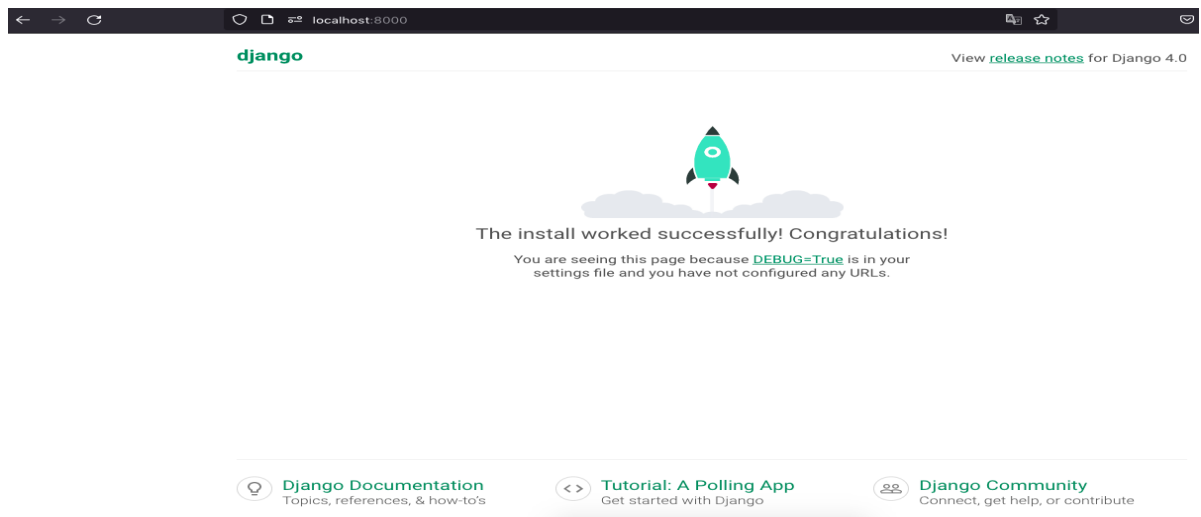
```
# python manage.py startapp music
```

A nuestra aplicación le hemos puesto el nombre de music, ya que nuestra aplicación se encargará de gestionar distintos tipos de categorías musicales. No será una aplicación completa ya que el objetivo de esta guía es al efecto de mostrar como crear, editar, actualizar y borrar datos de una base de datos desde una interfaz tal como lo hemos dicho en el resumen de esta.

Agregamos nuestra aplicación al proyecto, luego probemos si todo anda bien en nuestro servidor local

```
En el archivo Settings # INSTALLED_APPS = [  
    'core',  
]
```

```
# python manage.py runserver
```



Vemos que nuestro servidor está funcionando correctamente, por lo tanto hasta el momento no tenemos mayores inconvenientes. Paremos nuestro servidor y enfoquémonos en nuestra aplicación a fin de determinar la estructura de datos que utilizaremos.

CREANDO MODELOS EN DJANGO

Centrémonos primeramente en el archivo models, donde definiremos la estructura de los datos que almacenamos en nuestra base de datos (por cierto, utilizaremos la base de datos predeterminada de django). Obs: este proyecto es a efecto educativo y por eso hemos decidido hacerlo con un solo campo, el cual es el nombre de la categoría, sobre este campo aplicaremos el CRUD, es decir Crear, Recuperar, Actualizar y finalmente eliminar esas categorías. Creemos conveniente realizar esta guía de esta manera en razón de que todas las operaciones que realizaremos sobre este único campo, pueden fácilmente ser aplicadas a muchos campos, además nuestro objetivo es ser claros y sencillos lo cual no lo seríamos si trabajamos con muchos campos que confundiría muchas veces a los lectores. -

```
from django.db import models

class Category(models.Model):
    name = models.CharField(max_length=200, verbose_name='name', unique=True)

    def __str__(self) -> str:
        return self.name
```

Vayamos a nuestro archivo models.py el cual se encuentra dentro de nuestra aplicación, dentro del mismo, lo primero que necesariamente debemos de realizar es importar el módulo models.

models: es una clase que está incorporada en django y que utiliza para crear, recuperar, actualizar y eliminar datos de la base de datos.

Creemos una clase **Category**: que se implementa como una subclase de `models.Model`, y que puede incluir campos, métodos y metadatos.

Pasamos a definir posteriormente nuestro campo "name", el cual representará una columna dentro de nuestra tabla en la base de datos. Nuestro único campo es de tipo `models.CharField` (lo que significa que contendrá una cadena de texto, "caracteres alfanuméricos"). A nuestro campo "name", le hemos puesto tres argumentos, los cuales especifican a demás del tipo, cómo se guarda o cómo se puede usar.

- **max_length:**, el cual establece la longitud máxima de caracteres que podemos ingresar dentro de ese campo, en nuestro caso lo pusimos en 200.
- **verbose_name:** es la etiqueta que le especificamos.
- **unique:** como tercer argumento con el estado `true`, esto indica que el campo debe ser único en toda la tabla. En el caso que ingresemos un valor duplicado en un `unique=true`, este al intentar guardarlo en la base de datos nos arrojaría un error, que lo veremos más adelante.
- método **str(self)**, que indica como mostrar el objeto representado en un "string", es decir si no lo implementamos al momento de instanciar el objeto y devolver el mismo la representación del mismo sería `<main.Category object at 0x0000020B0787CA20>`, en cambio si lo implementamos, como en este caso, nos devolverá el nombre que ingresamos en el campo. Ejemplo "Django" en vez de lo anterior.

Una vez creado nuestro modelo, necesitamos ejecutar dos comandos necesarios: el "makemigrations" y el "migrate".

```
# python manage.py makemigrations
# python manage.py migrate
```

El `makemigrations` es el encargado de generar los comandos SQL para la aplicación preinstalada, mientras que el `migrate`, ejecuta estos comandos SQL en el archivo de la base de datos en nuestro caso el archivo predeterminado ubicado en la raíz de nuestro proyecto. Una vez ejecutados ambos comandos, en ese orden, se habrá creado una nueva tabla en la base de datos. Puedes comprobar yendo a tu proyecto > `app > migrations -> 0001_initial.py`.

ADMINISTRADOR PREDETERMINADO DE DJANGO

Vamos a renderizar nuestro modelo en el administrador predeterminado de Django, para ello vayamos a `app/admin.py` y coloquemos el siguiente código.

archivo `admin.py`

```
from django.contrib import admin

from .models import Category

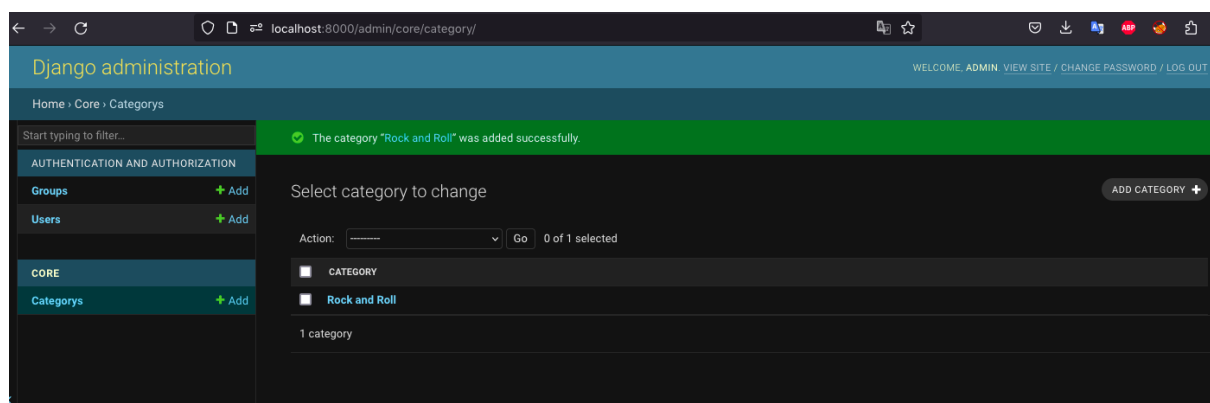
admin.site.register(Category)
```

Esto nos permitirá interactuar con nuestro modelo de la base de datos, es decir, crear, recuperar, actualizar y eliminar datos del mismo. Para ello creamos un password para poder acceder al panel de administración de Django.

```
# python manage.py createsuperuser
```

Aclaro que solo utilizaremos el administrador de Django, a manera de visualizar que nuestro modelo ha sido agregado correctamente, en el que podemos realizar ciertas acciones para comprobarlo.

En la siguiente imagen, se puede observar que se ha agregado al administrador de django nuestro modelo, en donde podemos realizar varias tareas desde el mismo.



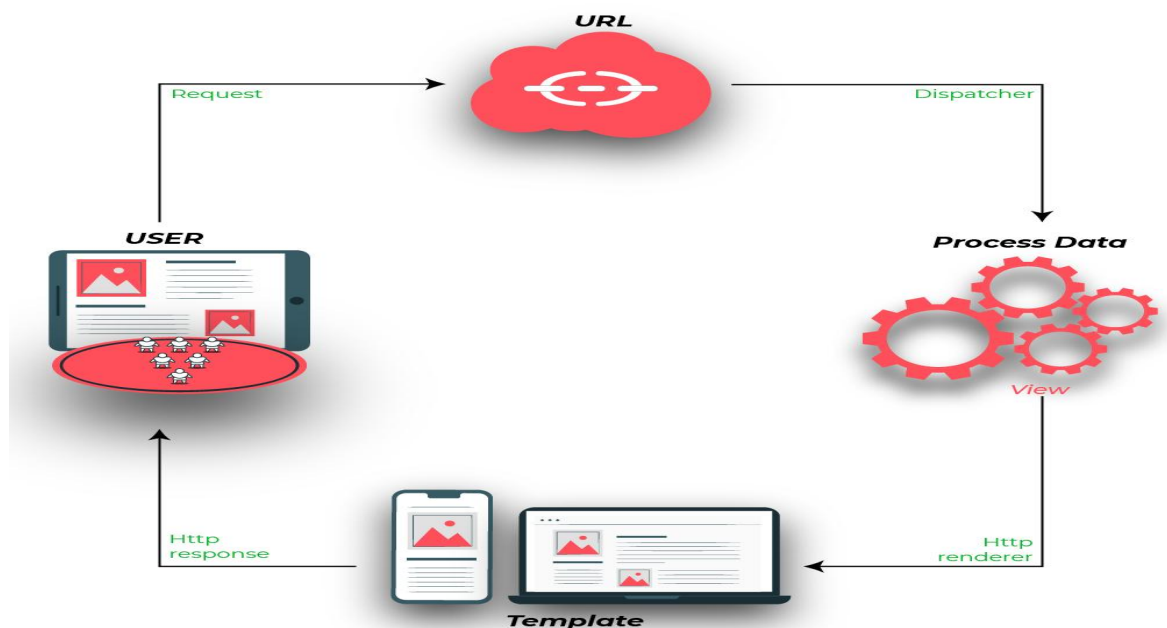
En este proyecto, nuestro objetivo es poder personalizar un panel de administrador a nuestro gusto y necesidades, utilizando varias herramientas.

CREACIÓN DE LAS VISTAS

Definido nuestro modelo, y hechas las migraciones correspondientes, nos enfocaremos en los "Views" o "Vistas", el cual es una de las principales partes de nuestra estructura MVT (models-views-templates).

A continuación crearemos y usaremos nuestra vista, considerando nuestro proyecto que contiene nuestra aplicación music, en el archivo core/views.py

Antes que nada desearía hacer un síntesis de lo que son la vista, a fin de tener una comprensión más acabada del mismo, según la documentación oficial de django es una función de python que toma solicitudes web y devuelve una respuesta web, prácticamente puede devolver cualquier cosa (XML, imagen, error 404 etc) y el código que lo contiene puede estar ubicado en cualquier parte, pero lo recomendado es hacerlo en un archivo views.py.



Fuente: <https://media.geeksforgeeks.org/wp-content/uploads/20200124153519/django-views.jpg>.

```
from django.shortcuts import render
from .models import Category
```

Importamos la función render desde django.shortcuts, este se encarga de renderizar una plantilla html, para ello necesita necesariamente que se le pase tres argumentos que debe retornar nuestra función.

- **request:** El objeto de solicitud utilizado para generar esta respuesta.
- **template_name:** El nombre completo de una plantilla para usar.
- **Un diccionario de valores** para agregar al contexto de la plantilla. Por defecto, este es un diccionario vacío. Si un valor en el diccionario es invocable, el view lo llamará justo antes de renderizar la plantilla, en nuestro caso estamos devolviendo los nombres de categoría existentes en nuestra base de datos.

```
from django.shortcuts import render
from .models import Category

def listCategory(request):
    template_name = "listCategory.html"
    cat = Category.objects.all()
    return render(request, template_name, {'cat':cat})
```

Es aquí, donde debemos detenernos y crear una carpeta dentro de nuestra aplicación con el objetivo de poner en ellas nuestras plantillas que utilizaremos para nuestro proyecto, para ello vayamos a nuestra aplicación music, y dentro de ella creemos una carpeta llamada "templates" y dentro de la misma un archivo html, que será el mismo utilizado en nuestra función listCategory() creado anteriormente.

ASIGNANDO UNA URL A LA VISTA

Una vez creada nuestra función listCategory y nuestra plantilla html, prácticamente estamos completando nuestra estructura MVT (models, views, templates), ahora necesitamos manejar la solicitudes, asignando una URL a esta vista para poder llamarlo desde nuestro navegador, para ello crearemos un archivo python dentro de nuestra aplicacion llamado urls.py y dentro del mismo crearemos el siguiente código. Para más información sobre los despachadores de urls, visiten el sitio oficial, donde encontrarán ejemplos muy sencillos y explicaciones exactas <https://docs.djangoproject.com/en/4.1/topics/http/urls/>.

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

urlpatterns = [
    path('listCategory/',listCategory, name='listCategory'),
]
```

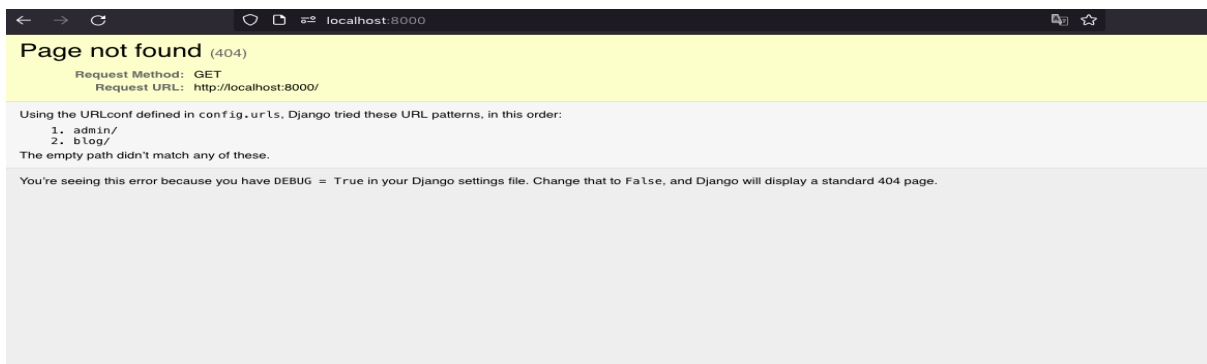
Realizado esto, debemos ir dentro de nuestra carpeta de configuración, al archivo al urls.py a fin de incluir nuestro archivo de configuración creado en nuestra aplicación de la siguiente manera.

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('core.urls')),
]
```

Lo que hacemos con `path('blog/', include('core.urls'))`, es incluir un nuevo módulo de configuración urls, "Cada vez que Django se encuentra [include\(\)](#), se corta cualquier parte de la URL que coincida hasta ese momento y envía el resto cadena a la URL incluida para su posterior procesamiento." . Fuente: <https://docs.djangoproject.com/en/4.1/topics/http/urls/>.

Guardemos, corramos el servidor y vayamos al local host <http://localhost:8000/>, donde django nos indica que no existe una coincidencia con la url pasada.



Si revisamos nuestro archivo url en el directorio de configuración, vemos que hemos ingresado como primer argumento en el path 'blog/', y como segundo argumento incluimos nuestro archivo urls ubicado en nuestra aplicación, donde habíamos puesto como primer argumento del path como 'listCategory/', por lo tanto a fin de haya una coincidencia con la url ingresada y django pueda llamar efectivamente a nuestra vista y devolvernos la plantilla html, necesitamos ingresar lo siguiente <http://localhost:8000/blog/listCategory>. Aquí django ha encontrado una coincidencia y ha llamado a nuestra vista "listCategory", el cual nos ha devuelto una plantilla html vacía hasta este momento.



DEVOLVIENDO UN OBJETO HttpResponse

Se acuerdan el tercer argumento de nuestro "return render(request, 'listCategory.html', {'cat':cat})" en el archivo views.py, formado por un diccionario el cual fue renderizado a nuestra plantilla html, el cual no solo tiene la capacidad para mostrar archivos estáticos sino también los datos de nuestra base de datos en específico, a través de etiquetas o variables dentro de llaves simples o llaves dobles según sea un condicional, un bucle o una variable. Probemos en nuestra plantilla mostrar lo que hay dentro del diccionario renderizado.

views.py

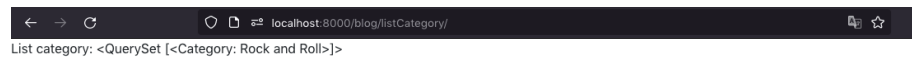
```
from django.shortcuts import render
from .models import Category

def listCategory(request):
    template_name = "listCategory.html"
    cat = Category.objects.all()
    return render(request, template_name, {'cat':cat})
```

listCategory.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>List Category</title>
</head>
<body>
    <p>List category: {{cat}}</p>
</body>
</html>
```

localhost:8000/blog/lisCategory/.



Vemos como se ha hecho la consulta y devuelto nuestro nombre de categoría puesto anteriormente, vayamos nuevamente a nuestro panel de administración predeterminado de django, y agreguemos unos nombres de categoría más.



localhost:8000/blog/listCategory/
List category: <QuerySet [<Category: Rock and Roll>, <Category: Blues>, <Category: Pop>, <Category: Trash Metal>]>

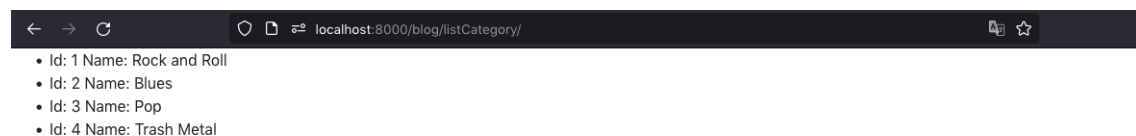
Las consultas o QuerySet son iterables, por lo tanto nosotros lo haremos a fin de ordenar en una lista nuestras categorías y posteriormente trabajarlo para nuestro objetivo. Para más información sobre los queryset, favor visitar el sitio oficial <https://docs.djangoproject.com/en/4.1/ref/models/querysets/>.

En nuestro listCategory.html

```
<ul>
  {% for categoria in cat %}
    <li>Id: {{categoria.id}} Name: {{categoria.name}}</li>
  {% endfor %}
</ul>
```

Aquí, lo que hacemos es iterar la consulta, en donde nos devuelve el id y el nombre de la categoría. Obs: al crear una nueva categoría, se crea un primary key que es su id, esto lo podemos ver dentro del archivo 0001_initial.py en la carpeta migrations de nuestra aplicación.

Actualizamos nuestro navegador.



localhost:8000/blog/listCategory/
• Id: 1 Name: Rock and Roll
• Id: 2 Name: Blues
• Id: 3 Name: Pop
• Id: 4 Name: Trash Metal

CREANDO TABLA DE DATOS CON DATATABLES

Una vez tengamos la capacidad de realizar la consulta a nuestra base de datos y renderizarlo a nuestra plantilla html, necesitamos organizar esos datos en el front-end a fin de que podamos mostrarlos de una manera sencilla y poder realizar más adelante lo que nos propusimos, el cual es tener la capacidad de crear, editar y eliminar categorías desde nuestra interfaz que estamos creando paso a paso. -

Para ello utilizaremos una herramienta muy útil llamada datatable, consistente en una extensión de jQuery que nos permite pintar tablas con paginado, búsqueda, ordenar por columnas, etc. Favor visitar la página oficial de DataTables en el siguiente link <https://datatables.net/>.

DataTables solo tiene una dependencia de biblioteca (otro software en el que se basa para funcionar): jQuery, para ello vayamos a la página oficial, descarguemos y guardemoslo, a modo de referencia., que es jquery?, según la documentación oficial: es una biblioteca de JavaScript rápida, pequeña y rica en funciones. Hace cosas como recorrido y manipulación de documentos HTML, manejo de eventos, animación y Ajax mucho más simple con una API fácil de usar que funciona en multitud de navegadores.

Creemos un directorio nuevo llamado static en nuestro directorio raíz, a fin de que contenga las librerías y herramientas que utilizaremos en nuestra aplicación, luego vayamos a nuestro archivo de configuración, y agreguemos STATICFILES_DIRS, esto a modo de contener nuestros archivos estáticos dentro de un solo directorio para todas nuestras aplicaciones que utilicemos en nuestro proyecto.

```
STATIC_URL = 'static/'

STATICFILES_DIRS = [
    BASE_DIR / "static",
]
```

Para más información sobre los STATICFILES_DIRS, visitar el sitio oficial de Django, donde encontrarán suficiente información con ejemplos claros y sencillos. <https://docs.djangoproject.com/en/4.1/howto/static-files/>.

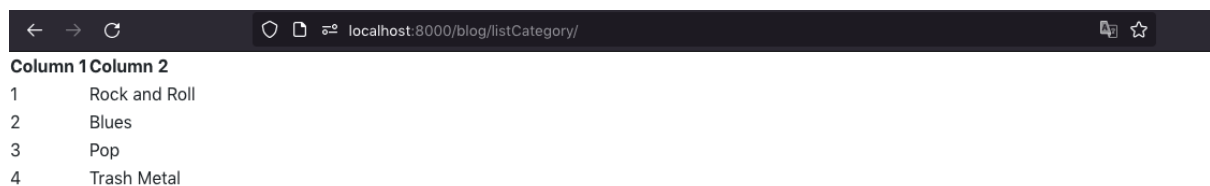
Luego dentro de nuestro archivo html, utilicemos la etiqueta "static" para construir la URL para la ruta relativa de nuestro archivo que contiene el jquery descargado.

```
{% load static %}
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>List Category</title>
    <!-- JQuery -->
    <script src="{% static 'lib/jquery-3.6.3.min.js'"></script>
</head>
```

Creemos una tabla dentro de nuestro archivo html, el código lo podemos encontrar dentro del sitio oficial de DataTables <https://datatables.net/manual/installation>., donde podemos encontrar ejemplos sobre cómo utilizar esta maravillosa herramienta.

```
<table id="table_id" class="display">
  <thead>
    <tr>
      <th>Column 1</th>
      <th>Column 2</th>
    </tr>
  </thead>
  <tbody>
    {% for category in cat %}
    <tr>
      <td>{{ category.id }}</td>
      <td>{{ category.name }}</td>
    </tr>
    {% endfor %}
  </tbody>
</table>
```

Actualizamos nuestra página y veremos nuestra tabla con los datos existentes en nuestra base de datos.



The screenshot shows a web browser window with the address bar displaying 'localhost:8000/blog/listCategory/'. The page content shows a table with two columns, 'Column 1' and 'Column 2', and four rows of data.

Column 1	Column 2
1	Rock and Roll
2	Blues
3	Pop
4	Trash Metal

Según los parámetros de la documentación de DataTables, tenemos una tabla válida para poder utilizarlo con esta herramienta, ahora nos queda implementar DataTable, primeramente debemos incluir los archivos fuentes en nuestra página, para ello vayamos a <https://datatables.net/download/> y descarguemos, luego incluye el archivo dentro del directorio static creado anteriormente.

archivo listCategory.html

```
<!-- DataTable -->
<link rel="stylesheet" type="text/css" href="{% static 'lib/DataTables/datatables.min.css'%}" />
<script type="text/javascript" src="{% static 'lib/DataTables/datatables.min.js'%}"></script>
```

Incluimos las rutas para que pueda cargarse conjuntamente con nuestro archivo html, una vez hecho esto, nuestra cabecera dentro de nuestro archivo html debería de lucir de la siguiente manera.

```

<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>List Category</title>

  <!-- JQuery -->
  <script type="text/javascript" src="{% static 'lib/jquery-3.6.3.min.js' %}"></script>

  <!-- DataTable -->
  <link rel="stylesheet" type="text/css" href="{% static 'lib/DataTables/datatables.min.css' %}" />
  <script type="text/javascript" src="{% static 'lib/DataTables/datatables.min.js' %}"></script>
</head>

```

Ahora iniciamos nuestra tabla de datos con DataTable, para ello dentro de nuestro documento, luego del body, creamos unas líneas de JavaScript y actualizamos. Obs: nuevamente el código para la inicialización lo podemos ver y estudiar dentro de la página oficial de DataTables donde encontrarán miles de ejemplos y formas de utilizar esta herramienta <https://datatables.net/examples/>.

```

<script type="application/javascript">
  $(document).ready( function () {
    $('#table_id').DataTable();
  });
</script>

```

Una vez actualizado la página podemos observar como se ha creado una tabla con DataTable donde contiene los datos de nuestra consulta.

show 10 entries	Search:
Column 1	Column 2
1	Rock and Roll
2	Blues
3	Pop
4	Trash Metal

¡Eso es todo! DataTables agrega ordenamiento, búsqueda, paginación e información a su tabla de manera predeterminada, brindando la capacidad de encontrar la información que desean lo más rápido posible. FUENTE: <https://datatables.net/manual/installation>.

TRABAJANDO CON AJAX

En esta sección utilizaremos Ajax para recuperar nuestros datos desde la base de datos, antes de empezar, crearemos un archivo javascript llamado funciones, donde incluiremos nuestro código apartado del archivo html, para organizarnos mejor. Como es una función propia, lo incluiremos dentro del directorio static creado previamente dentro de nuestra aplicación y agregaremos el directorio en nuestro archivo settings.

```
STATICFILES_DIRS = [  
    BASE_DIR / "static",  
    BASE_DIR / "core/static"  
]
```

Para más información sobre los STATICFILES_DIRS, visitar el sitio oficial de django donde podemos encontrar lo necesario para aprender lo suficiente para nuestro proyectos. <https://docs.djangoproject.com/en/4.1/howto/static-files/>.

Ahora vayamos a nuestro archivo funciones y creemos la solicitud de datos con Ajax para ser cargados directamente a nuestra tabla. Los datos de Ajax son cargados por DataTables simplemente usando las opciones que nos proporciona ajax, para esto nuestro código tanto en el archivo js funciones, listCategory.html y el views.py queda de la siguiente manera.

1.- Archivo listCategory.html dentro de nuestro directorio templates.

```
<body>  
{% csrf_token %}  
<table id="table_id" class="display">  
  <thead>  
    <tr>  
      <th>id</th>  
      <th>Name</th>  
    </tr>  
  </thead>  
</table>  
</body>
```

Aquí, lo primero que hicimos fue usar la protección CSRF de django, ya que realizaremos la solicitud de los datos a través del método POST, para ello usamos la etiqueta {% csrf_token %}, el cual será enviado junto con la solicitud de Ajax. Para más información sobre los CSRF de django visitar la documentación oficial de django en <https://docs.djangoproject.com/en/4.1/howto/csrf/>.

2.- Archivo funciones.js dentro del directorio static de nuestra aplicación.

```
$(document).ready( function () {  
    var csrftoken = document.getElementsByName('csrfmiddlewaretoken')[0].value  
    $('#table_id').DataTable( {  
        ajax:{  
            headers: {'X-CSRFToken': csrftoken},  
            url: window.location.pathname,  
            type: 'POST',  
            data: {  
                action: 'search',  
            },  
            dataSrc = ""  
        },  
        columns: [  
            {"data": [0]},  
            {"data": [1]},  
        ]  
    } );  
});
```

Aquí, utilizamos primeramente `$(document).ready()`, a fin de poder manipular correctamente nuestra página html hasta que el documento se haya cargado completamente, luego creo una variable `csrftoken` que es igual al valor que contiene nuestro `{% csrf_token %}`. Una vez hecho esto llamamos a `DataTable` y cuyos datos son recuperados a través de solicitud ajax, como se puede observar precedentemente.

2.1 - **headers** = en cada `XMLHttpRequest`, personalizamos el encabezado `X-CSRFToken` con el valor del token que obtuvimos con la variable `csrftoken`.

2.2 - **url** = `window.location.pathname`, nos devuelve la ruta y el nombre del archivo de la página actual, al cual realizará la petición.

2.3 - **data** = son los datos que enviamos junto a la petición, en este caso he enviado un action con el valor 'search', a fin de poder trabajarlo dentro de nuestro archivo `views.py`, que lo veremos más claramente a medida que vayamos terminando nuestra aplicación.

2.4 - **dataSrc** = se usa para decirle a `DataTables` donde está la matriz de datos en la estructura JSON. En nuestro caso le hemos puesto una cadena vacía como caso especial que le dice a `DataTables` que espere una matriz. En este caso no requiere configurar los datos de la columnas, esto se debe a que el valor predeterminado para las columnas es el índice las columnas `columns: [{"data": [0]}, {"data": [1]},]`, en nuestro caso solo nos devuelve una matriz con dos datos que se ubican en la posición 0 y 1.

3.- Archivo views.py dentro de nuestra aplicación.

```
from django.shortcuts import render
from .models import Category
from django.http import JsonResponse

def listCategory(request):
    data = {}
    if request.method == 'GET':
        template_name = 'listCategory.html'
        cat = Category.objects.all()
        return render(request, template_name)
    if request.method == "POST":
        try:
            action = request.POST['action']
            if action == 'search':
                data = list(Category.objects.all().values_list())
                return JsonResponse(data, safe=False)
        except Exception as e:
            data['error'] = str(e)
            return JsonResponse(data, safe=False)
```

Lo primero que realizamos es comparar las solicitudes realizadas a nuestra vista, comprobamos si el request.method corresponde a 'GET' o 'POST', al acceder a nuestra url listCategory/ a través del método 'GET' se renderiza a nuestro template_name.

Anteriormente hablamos sobre \$(document).ready(), una vez cargado completamente nuestra pagina, se llama a DataTable y se realiza la petición 'POST' con Ajax, esto llama nuevamente a nuestra vista, en donde lo manejamos con los condicionales, en este caso ingresa al segundo condicional, realizamos un try que nos permite probar el bloque de código y manejar cualquier tipo de error que pudiera suceder al tratar de obtener el valor de request.POST['action'] o al momento de hacer la consulta con Category.objects.all(). Luego volvemos a agregar un condicional en donde prueba si la variable action es igual a 'search', si esto es true entonces ingresa en el bloque y dentro de la variable data obtiene las categorías existentes, pero sin antes convertir los datos y enviarlos con JsonResponse que transforma los datos que le pasa en una cadena JSON y establece el encabezado HTTP del tipo de contenido en application/json.

Luego en caso de que ocurra algún error dentro del código try, controlamos el error con except Exception as e, que nos devolverá un mensaje con el tipo de error ocurrido, lo agregamos a data['error'] y lo retornamos con JsonResponse.

Ahora bien, hagamos correr nuestro servidor y verificar que todo salió bien.

Show 10 entries		Search: <input type="text"/>	
id	*	Name	
1		Rock and Roll	
2		Blues	
3		Pop	
4		Trash Metal	
Showing 1 to 4 of 4 entries		Previous	1 Next

Showing 1 to 4 of 4 entries

id	Name
1	Rock and Roll
2	Blues
3	Pop
4	Trash Metal

Showing 1 to 4 of 4 entries

Previous 1 Next

Inspector Console Depurador Red Editor de estilos Rendimiento Memoria Almacenamiento Accesibilidad Aplicación Adblock Plus

Filtrar salida

Cabeceras Cookies **Solicitud** Respuesta Tiempos Traza de la pila

Filtrar los parámetros de la petición

Datos de formulario

action: "search"

Sin procesar

En esta captura se puede observar, que se ha cargado correctamente el DataTables, aquí vamos a inspeccionar el documento html, que se ha enviado en la solicitud el dato que le hemos pasado con "action" y su valor 'search'.

Showing 1 to 4 of 4 entries

id	Name
1	Rock and Roll
2	Blues
3	Pop
4	Trash Metal

Showing 1 to 4 of 4 entries

Previous 1 Next

Inspector Console Depurador Red Editor de estilos Rendimiento Memoria Almacenamiento Accesibilidad Aplicación Adblock Plus

Filtrar salida

Cabeceras Cookies Solicitud **Respuesta** Tiempos Traza de la pila

Filtrar propiedades

JSON

```
[{"id": 1, "name": "Rock and Roll"}, {"id": 2, "name": "Blues"}, {"id": 3, "name": "Pop"}, {"id": 4, "name": "Trash Metal"}]
```

Sin procesar

Y como respuesta una matriz con los datos provenientes de nuestra base de datos, cargado previamente a través del administrador predeterminado de Django.

CREANDO UN NUEVO REGISTRO

Una vez que hemos recuperado nuestros datos de la base de datos con Ajax y mostrado a través de DataTables, en esta sección veremos como poder cargar un registro desde nuestra página listCategory.html, utilizando los modals de bootstrap, rellenando un formulario en nuestro front-end y enviarlos a nuestro back-end para que sea procesado, validado y creado un nuevo registro que será mostrado en nuestra tabla.

Para ello primeramente vayamos a la pagina oficial de bootstrap <https://getbootstrap.com/docs/4.6/getting-started/download/>, descarguemolo y guardemolo dentro de static en nuestro directorio raiz en la carpeta lib, agreguemolo en nuestro archivo html de la siguiente manera.

```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>List Category</title>

  <!-- JQuery -->
  <script type="text/javascript" src="{% static 'lib/jquery-3.6.3.min.js' %}"></script>

  <!-- DataTable -->
  <link rel="stylesheet" type="text/css" href="{% static 'lib/DataTables/datatables.min.css' %}" />
  <script type="text/javascript" src="{% static 'lib/DataTables/datatables.min.js' %}"></script>

  <!-- Bootstrap v4-->
  <link rel="stylesheet" href="{% static 'lib/bootstrap-4.6.2-dist/css/bootstrap.min.css' %}">
  <script src="{% static 'lib/bootstrap-4.6.2-dist/js/bootstrap.min.js' %}"></script>

  <!-- Funciones propias -->
  <script type="text/javascript" src="{% static 'funciones.js' %}"></script>
</head>
```

En este tutorial solo nos abocaremos a la tarea de mostrar el funcionamiento del CRUD, no así los estilos en el front-end, por lo que queda a criterio y gusto de vosotros el darles los estilos que les parezca más conveniente. En este enlace pueden personalizar DataTables con bootstrap <https://datatables.net/examples/styling/bootstrap4.html>.

Una vez incorporado bootstrap a nuestro archivo html, vayamos a <https://getbootstrap.com/docs/4.6/components/modal/>, en dicho sitio veremos los ejemplos de modals que nos serán de utilidad para poder realizar la tarea que nos propusimos en esta sección.

Archivo listCategory.html

```
<body>
  {% csrf_token %}
  <div class="container-fluid m-2">
    <table id="table_id" class="table table-striped table-bordered">
      <thead>
        <tr>
          <th>id</th>
          <th>Name</th>
        </tr>
      </thead>
    </table>
    <button class="btn btn-primary" id="btnCreateCategory">Register Category</button>
    <button class="btn btn-success" id="buttonUpdate">Update</button>
  </div>

</body>
<div class="modal fade" id="modal_category" tabindex="-1" aria-labelledby="exampleModalLabel"
aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="exampleModalLabel">Modal title</h5>
        <button type="button" class="close" data-dismiss="modal" aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
      </div>
      <div class="modal-body">
        ...
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>
        <button type="button" class="btn btn-primary">Save changes</button>
      </div>
    </div>
  </div>
</div>

</div>
```

Agregamos a nuestro archivo html, la siguiente línea copiada de los ejemplos de modals, en el que se puede observar que lo hemos puesto al terminar el body. Lo importante y lo que utilizaremos para llamar a nuestro modal será el (id="modal_category"), como así también el button Register Category le hemos puesto un identificador "btnCreateCategory", a fin de que con el evento click podemos llamar a aquel. Dentro de nuestro archivo funciones.js escribimos la siguiente línea de código.

```
$(document).ready( function () {
  var csrftoken = document.getElementsByName('csrfmiddlewaretoken')[0].value
  $('#table_id').DataTable( {
    ajax:{
      headers: {'X-CSRFToken': csrftoken},
      url: window.location.pathname,
      type: 'POST',
      data: {
        action: 'search',
      },
    },
```

```

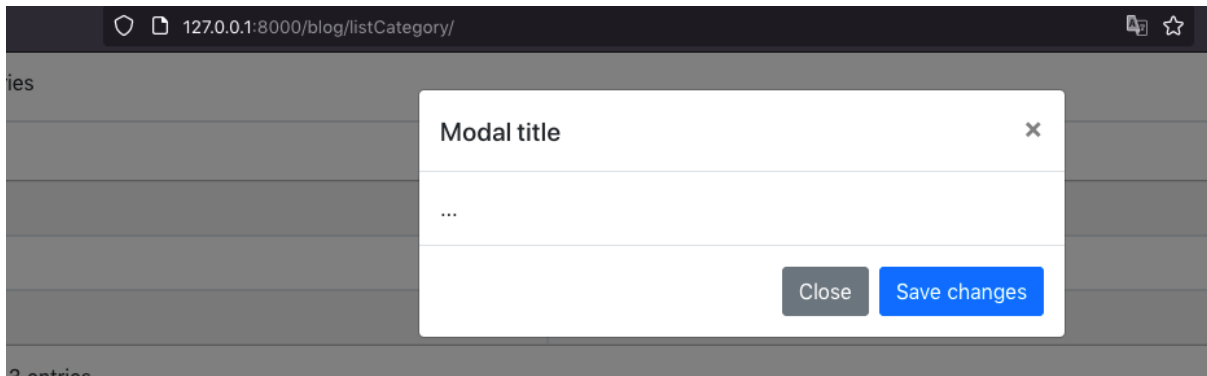
        dataSrc: ""
    },
    columns: [
        {"data": [0]},
        {"data": [1]},
    ]
});

$('#buttonUpdate').on('click', function(){
    $('#table_id').DataTable().ajax.reload();
});

$('#btnCreateCategory').on('click', function(){
    $('#modal_category').modal('show');
});
});

```

Con `$('#btnCreateCategory')` tomamos el evento click y disparamos la función que a su vez toma nuestro modals y lo muestra con `$('#modal_category').modal('show')`, guardémoslo y hacemos click en el botón Register Category.



Una vez hecho esto, tendremos que poner en nuestra ventana modals de bootstrap un formulario, en nuestro caso solo esta compuesto con un campo el cual es el nombre de la categoría, el formulario propiamente dicho es un punto de entrada que una vez completado es enviado a nuestra back-end para ser procesado, validado y guardado en su caso.

Para ello creamos un archivo `forms.py` dentro del directorio de nuestra aplicación, en el cual realizaremos el siguiente código.

```

from django import forms
from .models import Category

class CategoryForm(forms.ModelForm):
    class Meta:
        model = Category
        fields = "__all__"

    widgets = {
        'name': forms.TextInput(attrs={
            'class': 'form-control',
            'placeholder': 'enter a new category',
            'autocomplete': 'off',
        })
    }

```

Aquí utilizaremos un formulario basado en nuestro modelo Category, esto implica que se utilizan las características de nuestro modelo para crear el formulario y así ahorrar bastante tiempo y código. La documentacion oficial lo pueden encontrar en <https://docs.djangoproject.com/en/4.1/topics/forms/modelforms/#modelform>.

Luego debemos importarlo en nuestro archivo views.py a fin de poder renderizarlo en nuestra vista de la siguiente manera.

```
from django.shortcuts import render
from .models import Category
from django.http import JsonResponse
from .forms import CategoryForm

def listCategory(request):
    data = {}
    if request.method == 'GET':
        template_name = 'listCategory.html'
        cat = Category.objects.all()
        form = CategoryForm()

        return render(
            request,
            template_name,
            {
                'form': form,
            })

    if request.method == "POST":
        try:
            action = request.POST['action']
            if action == 'search':
                data = list(Category.objects.all().values_list())
                return JsonResponse(data, safe=False)
        except Exception as e:
            data['error'] = str(e)
            return JsonResponse(data, safe=False)
```

Como se puede observar dentro de la variable "form" colocamos nuestro formulario compuesto de un campo "name", el cual es renderizado a nuestro archivo html a fin de poder utilizarlo dentro de nuestro modal.

```
<div class="modal fade" id="modal_category" tabindex="-1" aria-labelledby="exampleModalLabel"
aria-hidden="true">
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h5 class="modal-title" id="modal_title"></h5>
        <button type="button" class="close" data-dismiss="modal" aria-label="Close">
          <span aria-hidden="true">&times;</span>
        </button>
      </div>
      <div class="form-group">
        <form method="POST" action=".">
          {% csrf_token %}
          <div class="modal-body">
            <label>Category Name</label>
            {{form.name}}
          </div>
        </form>
      </div>
    </div>
  </div>
```



```

<div class="modal-footer">
  <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>
  <button type="submit" class="btn btn-primary">Save changes</button>
</div>
</form>
</div>
</div>
</div>
</div>

```

En nuestro archivo html, en el modals que se muestra una vez que presionamos el botón creamos una etiqueta dentro del cual, agregamos el csrf_token y nuestro formulario {{ from.name }} y la etiqueta label, dentro del bloque form, asimismo el botón "save changes", lo convertimos al tipo submit para poder enviar nuestro formulario a nuestro back-end.

Luego en nuestro archivo funciones.js agregamos las siguientes líneas de código.

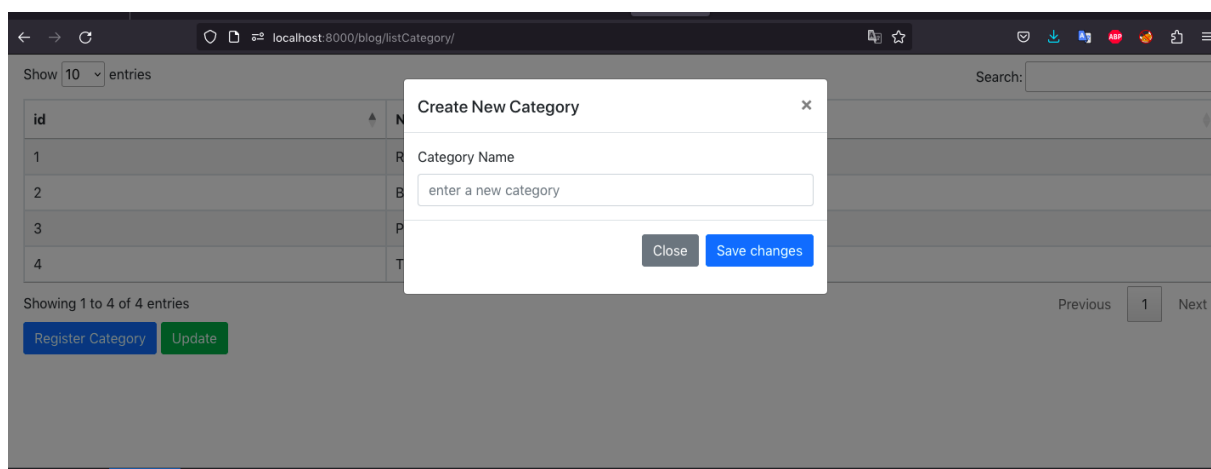
```

$('#btnCreateCategory').on('click', function(){
  $('#modal_category').modal('show');
  $('#modal_category form')[0].reset();
  $('#modal_title').text('Create New Category')
});

```

Aquí, agregamos un \$('#modal_category form')[0].reset(); lo que hace es eliminar cualquier registro dentro de nuestro formulario, esto en razón de que al escribir dentro de un formulario ubicado en un modals y cerrarlo posteriormente, el campo relleno permanece con lo cargado anteriormente, si es que lo hacemos, por lo que con esta función lo que hacemos es limpiar dicho campo. Luego con \$('#modal_title').text('Create New Category') agregamos un título a nuestro modal que variara según estemos creando un nuevo registro o editando un registro, esto lo veremos posteriormente. -

Vayamos a nuestra url <http://127.0.0.1:8000/blog/listCategory/> y presionemos el botón Register Category, el resultado sería esto.



Nos aparece el modals de bootstrap, con un campo de formulario en el que podemos enviar con el botón "save changes", el cual veremos en la próxima sección.

ENVIANDO LOS DATOS DEL FORMULARIO

En esta sección enviaremos el formulario creado a nuestro back-end a fin de que lo procese, valide y en su caso guarde. En nuestro archivo funciones.js creamos lo siguiente.

```
$( 'form' ).on( 'submit', function( e ) {  
    e.preventDefault()  
    var data = $( 'form' ).serializeArray()  
    var csrftoken = data[0].value  
    $.ajax(  
        {  
            headers: { 'X-CSRFToken': csrftoken },  
            url : window.location.pathname,  
            type: "POST",  
            data : data,  
        }  
    ).done( function( data ) {  
        if( !data[ 'error' ] ) {  
            $( '#modal_category' ).modal( 'hide' );  
            $( '#table_id' ).DataTable().ajax.reload();  
            return false  
        }  
        else {  
            alert( data )  
        }  
    } )  
    .fail( function( data ) {  
        alert( "error" );  
    } )  
    .always( function( data ) {  
    } );  
});
```

Con `$('form').on('submit', function(e) {` lo que hacemos es capturar el evento submit, una vez rellenado nuestro campo name de nuestra categoría nueva, luego con `e.preventDefault()`, lo que hacemos es cancelar el evento submit y evitar que se envíe el formulario a nuestro back-end, esto lo hacemos necesariamente para controlar el evento y poder enviar nuestro formulario a través de ajax. Con `var data = $('form').serializeArray()` lo que hacemos es tomar los valores de nuestro formulario y crear un array con ellos, esto lo podemos constatar si realizamos un `console.log(data)` y `var csrftoken = data[0].value` que toma el valor del `{% csrf_token %}` para enviarlo conjuntamente con la petición y enviar que nos nos arroje un error en el lado del servidor.

Posteriormente con `$.ajax({})` realizamos la petición a nuestro servidor, enviando los datos necesarios para que lo procese, al igual que lo hicimos con la petición en nuestro DataTables, enviamos `{ headers: { 'X-CSRFToken': csrftoken }, url : window.location.pathname, type: "POST", data : data, }`, donde con `data: data`, estamos enviando nuestro formulario. Pueden leer más visitando el siguiente sitio <http://www.falconmasters.com/jquery/peticiones-ajax-jquery/>.

En nuestro archivo html, específicamente dentro del `<form></form>` agregamos un input de tipo hidden, es decir oculto y con el nombre de action que tiene un valor vacío por el momento, fijémonos.

```

<form method="POST" action=".">
    {% csrf_token %}
    <input type="hidden" name="action" value="">
    <div class="modal-body">
        <label>Category Name</label>
        {{form.name}}
    </div>
    <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>
        <button type="submit" class="btn btn-primary">Save changes</button>
    </div>
</form>

```

Agregamos la siguiente línea, `<input type="hidden" name="action" value="">`, por qué motivo?, este input es enviado junto con nuestro formulario, por lo tanto, recuerdan que en nuestra solicitud en el DataTables con ajax habíamos enviado un input oculto con el nombre de action y con un valor de "search", esto a fin de que dentro de nuestra vista al llegar la solicitud comparemos el valor de action, si corresponde a "search", nos devolverá la lista completa de registros existentes en nuestra base de datos, si corresponde a otro valor, como por ejemplo "create", "edit" o "delete", realizaría otra acción creando un nuevo registro, editando un registro o eliminándolo. Rellenamos nuestro formulario y hacemos click en "save changes", ahora fijémonos en la consola cuáles son los datos enviados a nuestro servidor.

```

csrfmiddlewaretoken
"GpvqDLGfbUyFN5F0Q2hFEHP2oNQdrHbHnDU69S3lOoieyeBJX3MYPXwdgBxtWLoS"
action  "create"
name    "Django"

```

Vemos que se ha enviado estos datos, el action nos servirá como dijimos anteriormente para verificar que proceso realizaremos en el back-end, el name contiene el nuevo registro que será procesado, verificado y en su caso guardado, decimos en su caso ya que nuestro modelo name solo permite la existencia de registros únicos no repetidos, esto lo veremos más claramente en seguidamente. -

```

def listCategory(request):
    data = {}
    if request.method == 'GET':...

    if request.method == "POST":
        try:
            action = request.POST['action']
            if action == 'search':
                data = list(Category.objects.all().values_list())
                return JsonResponse(data, safe=False)
            elif action == 'create':
                form = CategoryForm(request.POST)
                if form.is_valid():
                    form.save()
                    return JsonResponse(data, safe=False)
            else:
                data['error'] = f"Category {request.POST['name']} already exists"
                return JsonResponse(data, safe=False)
        except Exception as e:

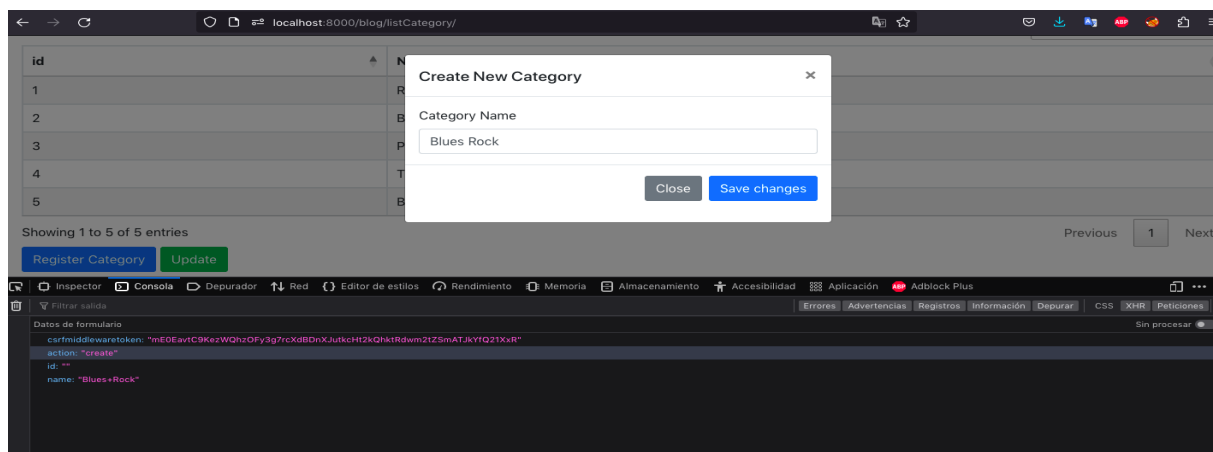
```

```
data['error'] = str(e)
return JsonResponse(data, safe=False)
```

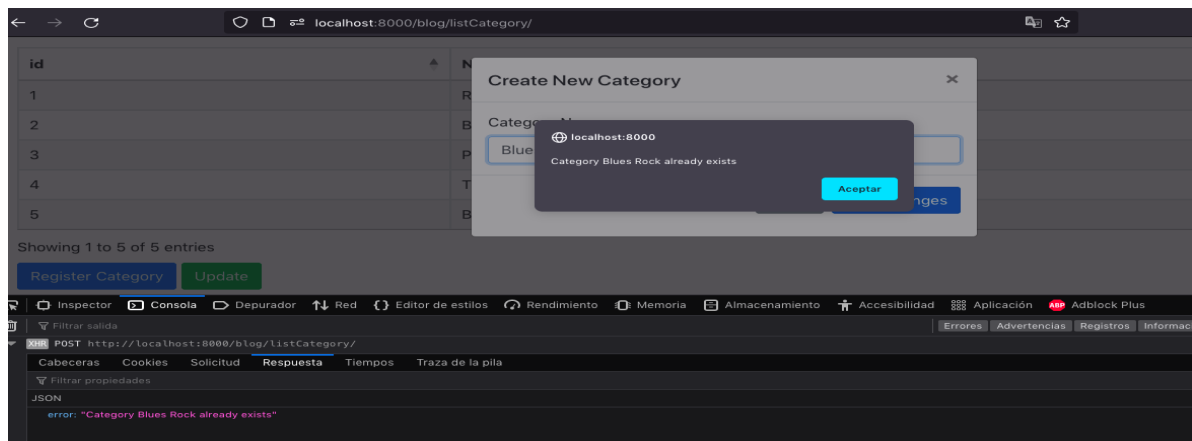
Aquí en nuestra vista, lo que primero hacemos es comparar si action corresponde con "create", en caso de que sea true, dentro de la variable form, llamamos a CategoryForm y le pasamos como parametro el request.POST que contiene nuestro formulario, seguidamente verificamos si el formulario es válido con "is_valid()", si lo fuere guarda el registro en la base de datos con form.save() y retorna un JsonResponse(data, safe=False), con un data vacío, en caso contrario si "is_valid() " es False, guardamos dentro de un diccionario data el error en nuestro caso sería la existencia del mismo registro y lo devolvemos con JsonResponse para que podamos mostrarlo en nuestra interfaz.

En nuestro caso al registrar un nombre válido (es decir no repetido en nuestra base de datos), lo carga y se actualiza nuestra tabla con \$('#table_id').DataTable().ajax.reload(); dentro de nuestro archivo funciones.js.

Es importante aclarar que hemos puesto un botón update, a fin de poder actualizar nuestra tabla sin necesidad de recargar toda la página, esto lo realiza tomando el evento click de dicho botón y recargando la tabla con \$('#table_id').DataTable().ajax.reload(). -



Vemos que se ha registrado correctamente nuestra categoría, con un nombre y un id, ahora carguemos un registro existente a fin de verificar si nos arroja el error que hemos configurado en nuestro back-end.



Podemos ver como se ha generado un alert con un mensaje "Category Django already exists", esto lo podemos corroborar en nuestro views.py en `data['error'] = f"Category {request.POST['name']} already exists"`, como consecuencia de que el registro "Django" ya existe en la base de datos, y lo colocamos en el diccionario data el cual lo retornamos con `JsonResponse(data, safe=False)`. En nuestro archivo funciones.js se ve de la siguiente manera.

```
.done(function(data) {
  if(!data['error']){
    $('#modal_category').modal('hide');
    $('#table_id').DataTable().ajax.reload();
    return false
  }else{
    alert(data.error)
  }
})
```

Con `.done(function())`, lo que hacemos es verificar si el data retornado no contiene un 'error', si es true, ocultamos el modal y actualizamos nuestro DataTables, en caso contrario si contiene un 'error', nos arroja un `alert(data.error)` con el mensaje que le hemos pasado en nuestro back-end. Este mensaje lo podemos personalizar el mensaje de alerta visiten el sitio web <https://sweetalert2.github.io/>, es muy fácil de implementar, por cuestiones de tiempo en esta sección no lo mostraremos.

RENDERIZADO DE DATOS EN DATATABLES

Hasta ahora hemos cargado nuestro DataTables con una petición asíncrona con Ajax a nuestro back-end, así también hemos creado un registro utilizando formulario ubicado en un modals con bootstrap y la petición lo hemos realizado con Ajax a nuestro servidor para que lo procese, valide, guarde en su caso o arroje un mensaje de error.

Nuestro CRUD (Create, Read, Update, Delete) se va completando de a poco, hemos realizado los dos primeros (Create y Read), nos faltaría (Update y Delete). Tanto el editar como el eliminar una categoría (lo los registros que existen en nuestra base de datos y son muestran en la tabla con Datatables), necesitamos seleccionar dicho registro con el que podamos editarlo y eliminarlo posteriormente, para ello utilizaremos el renderizado de los datos que son mostrado dentro de nuestro tabla a fin de que nos dé la posibilidad de incorporar dos botones a cada fila con las opciones de editar y eliminar.

¿Qué es el renderizado? Vayamos a la página oficial de DataTables en <https://datatables.net/reference/option/columns.render> en el que podremos ver ejemplos claros como utilizar el renderizado de datos que son utilizados en nuestra tabla.

Hasta el momento nuestra tabla consta de 2 columnas, una para el id y otra para el nombre de nuestra categoría, nos faltaría agregar una columna más, en el que nos dé la opción de seleccionar el registro ubicado en la fila y editarlo o eliminarlo en su caso, por lo tanto nuestra tabla quedaría de la siguiente manera.

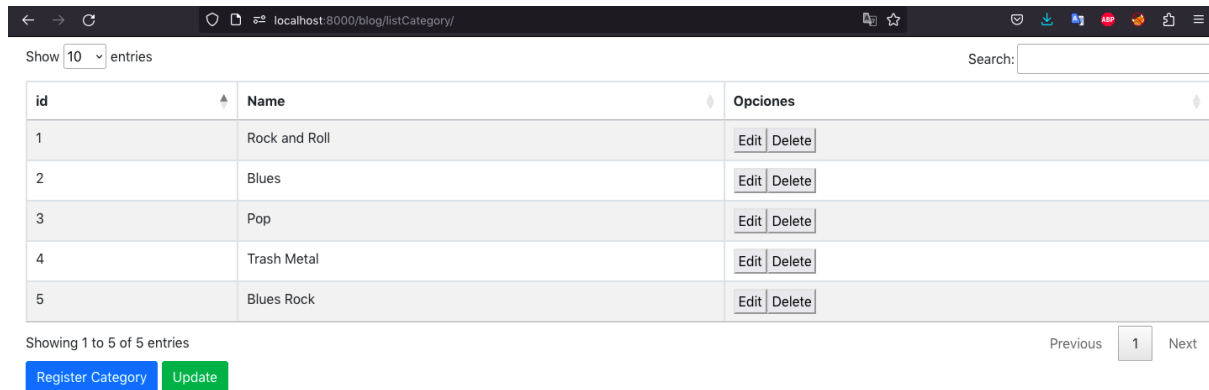
```
<thead>
<tr>
  <th>id</th>
  <th>Name</th>
  <th>Opciones</th>
</tr>
</thead>
```

Le hemos agregado una columna llamada opciones, en el cual debemos procesar y modificar los datos que se mostrarán en él, para ello como dijimos anteriormente utilizaremos el poder de DataTables y el renderizado de columnas, por lo tanto vayamos a nuestro archivo funciones.js y modifiquemoslo para que podamos renderizar a nuestra tabla los botones editar y eliminar.

```
$('#table_id').DataTable( {
  ajax:{
    headers: {'X-CSRFToken': csrftoken},
    url: window.location.pathname,
    type: 'POST',
    data: {
      action: 'search',
    },
    dataSrc: ""
  },
  columns: [
    {"data": [0]},
    {"data": [1]},
    {"data": [1]},
  ],
  "columnDefs":[
    {
      "targets": 2,
      "render": function(data,type,row){
        var button = "<button>Edit</button>"
        button += "<button>Delete</button>"
        return button
      }
    }
  ]
});
```

Vemos como en la opción columns hemos agregado un {"data": [1]}, esto es en razón de que hemos agregado una columna a nuestro datatable, por lo tanto debemos rellenarlo con un dato necesariamente, en este caso es el mismo que el nombre de categoría, pero aquí viene lo interesante, más abajo le hemos agregado un columnDefs: [], con esto podemos procesar y modificar los datos que se muestran en cada columna de nuestra tabla, pueden observar como en "targets" hemos puesto el numero 2, esto porque la

columnas es una matriz [id,name,opciones] en el que el id se encuentra en la posición número 0 de la matriz, el name en la posición número 1 y opciones en la posición número 2, es por esto que el objetivo o targets al cual apuntamos es a la columna opciones, luego realizamos un render con un función con tres parametros (data, type, row), dentro de la función creamos una variable button el cual contiene un texto plano "Edit", el cual una vez retornado se convierte en un botón en nuestra tabla. Vayamos a nuestro navegador y actualizamos la página, se tendría que ver de la siguiente manera.



The screenshot shows a web browser at localhost:8000/blog/listCategory/. The page displays a table with 5 entries. The table has three columns: 'id', 'Name', and 'Opciones'. The 'Opciones' column contains 'Edit' and 'Delete' buttons for each row. Below the table, there are buttons for 'Register Category' and 'Update', and pagination controls showing 'Previous', '1', and 'Next'.

id	Name	Opciones
1	Rock and Roll	Edit Delete
2	Blues	Edit Delete
3	Pop	Edit Delete
4	Trash Metal	Edit Delete
5	Blues Rock	Edit Delete

Showing 1 to 5 of 5 entries

Previous 1 Next

Register Category Update

Hemos renderizado los datos que se muestran en la columna opciones, colocando dos botones "edit" y "delete", que nos servirá para editar un registro o eliminarlo en su caso. Vuelvo a reiterar que no me centraré en la parte estética de nuestra página sino en su funcionalidad, al realizar el renderizado de los botones pueden agregar las clases de bootstrap como así también los iconos de la siguiente pagina de bootstrap <https://icons.getbootstrap.com/>, una vez agregado a su archivo html lo pueden utilizar, en la mencionada página pueden encontrar los iconos que más le gusten y utilizarlos dentro de su proyecto con mucha rapidez y facilidad.

EDITANDO UN REGISTRO

Capturando el evento click del botón edit renderizado y recuperando los datos del registro seleccionado

Una vez renderizado nuestros botones de editar y eliminar dentro de nuestra tabla, vamos a comenzar la tarea de editar un registro al pulsar click en el botón edit, que abra una ventana modals y que muestre los datos de la fila seleccionada. Para ello necesitamos tomar el evento click del botón edit, buscar los datos en la fila correspondiente, guardarlos en una variable, mostrarlo en la ventana modal y poder editarlo, una vez editado realizar un submit que enviará el nuevo dato, si hubo algún cambio a nuestro back-end para que en las views.py lo procese, lo verifique y lo guarde en su caso.

Primeramente tenemos que capturar el evento click en el botón edit de nuestra DataTables, para ello necesitamos agregarle al botón un atributo para que nos dé la posibilidad de capturar el evento al hacer click en el mismo, para ello necesitamos realizar lo siguiente dentro de nuestro archivo funciones.js, en el renderizado de los botones.

```

$(document).ready( function () {
var table;
var csrftoken = document.getElementsByName('csrfmiddlewaretoken')[0].value
table = $('#table_id').DataTable( {
ajax:{
headers: {'X-CSRFToken': csrftoken},
url: window.location.pathname,
type: 'POST',
data: {
action: 'search',
},
dataSrc: ""
},
columns: [
{"data": [0]},
{"data": [1]},
{"data": [1]},
],
"columnDefs":[
{
"targets": 2,
"render": function(data,type,row){
var button = "<button rel='edit'>Edit</button>"
button += "<button>Delete</button>"
return button
}
}
]
});

```

Como se daran cuenta hemos creado una variable table que es igual a nuestra tabla de DataTables, esto lo utilizaremos posteriormente, en la parte inferior del código en el columnDefs hemos puesto en la variable button un rel='edit', esto a fin de capturar el evento click, no le hemos puesto un identificador id='edit' en razón de que los ids son únicos y no nos funcionaria. Una vez hecho esto guardamos y ejecutamos a fin de inspeccionar el elemento y verificar que se ha renderizado correctamente.

Ahora bien, necesitamos capturar el evento click y los datos que corresponden a dicha fila, para ello dentro de nuestro archivo funciones.js agreguemos las siguientes líneas de código, que trataré de explicar lo más sencillo posible. -

Archivo funciones.js

```

$('#table_id tbody').on('click', 'button[rel="edit"]', 'tr', function(){
var data = table.row($(this).parents('tr')).data();
console.log(data);
})

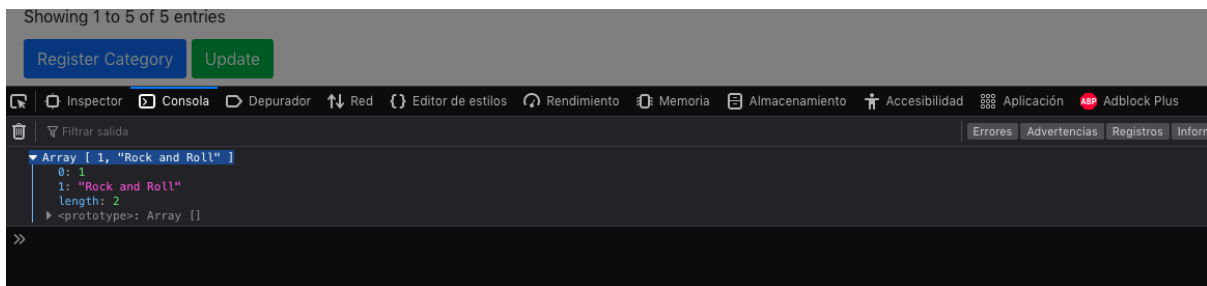
```

Con el objetivo de tomar el evento click en el boton editar de cada fila, necesitamos precisar el evento que queremos tomar para ello apuntamos al id de nuestra tabla especificamente en el tbody, en donde estan las filas con los datos de nuestro registro, al realizar click, en el boton con la relacion edit, en la tr o fila de tabla se dispara una funcion, en donde con var data es igual a los datos existentes en dicha fila, esto lo podemos

encontrar como ejemplo en https://datatables.net/examples/ajax/null_data_source.html, luego hacemos un console.log de data para ver que nos muestra al presionar el boton edit.

Acuerdense que para esto necesitamos crear una variable que sea igual a nuestra tabla tal como lo he mencionado precedentemente. -

A nuestra variable table.row() le pasamos la posición actual de la fila con \$(this).parents('tr') y con table.row().data(); obtenemos los datos de dicha fila y lo ponemos en la variable data.

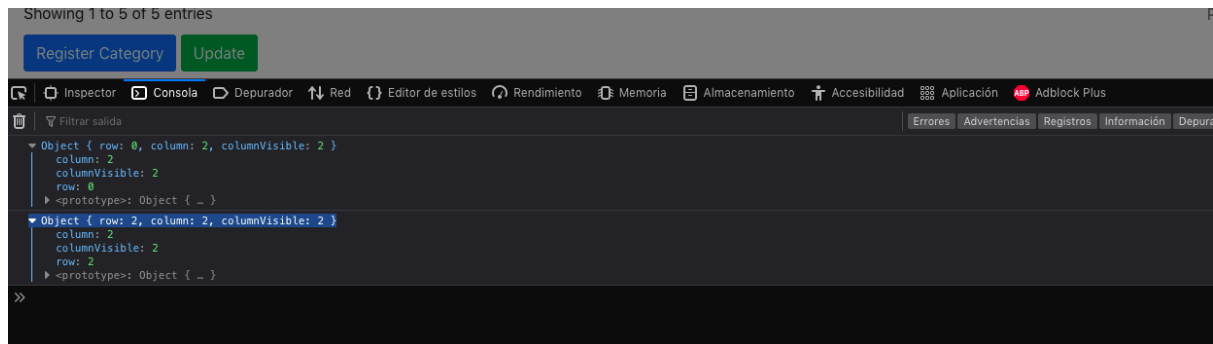


Aquí podemos observar cómo hemos recuperado los datos de nuestra tabla al hacer click en el botón click en una fila determinada, lo que tranquilamente ya podemos manipular para utilizarlo en nuestro modals. Pero aquí suele existir un problema al obtener la fila actual, en razón de que parents('tr') obtiene los padres de la fila seleccionada, en caso de que haya un cambio en las propiedades de la tabla como ser por ejemplo cuando lo hacemos responsive, se van creando otros elementos en la tabla por consiguiente no podrá encontrar los parents('tr'), es por ello que existe una solución mucho mejor, y lo podemos encontrar en [https://datatables.net/reference/api/cell\(\).index\(\)](https://datatables.net/reference/api/cell().index())

Con la función cell().index() podemos obtener el índice de la fila, columna y columna visible lo que a nosotros nos interesa es obtener solo el índice de la fila, tal como lo hemos obtenido con el método anterior para ello nuestro código debe quedar de la siguiente manera.

```
$('#table_id tbody').on('click', 'button[rel="edit"]', 'tr', function(){
    var td = $(this).closest('td, li');
    var tr = table.cell(td).index()
    console.log(evento);
    console.log(tr);
});
```

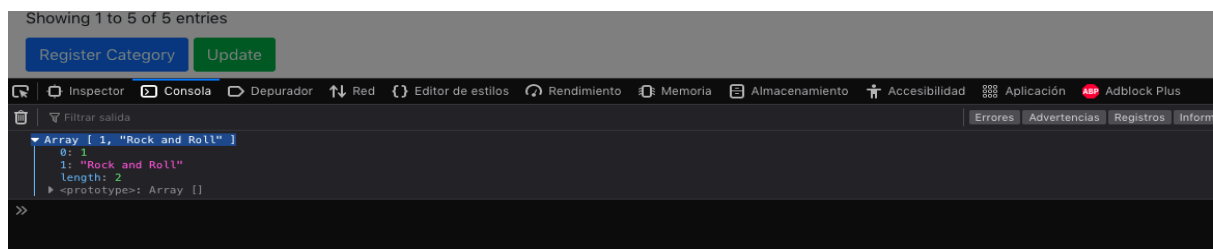
Aquí lo que hacemos primeramente con la variable td es obtener los datos de una celda de datos creados con la etiqueta , luego en la variable tr obtenemos el índice a través del método cell pasandole nuestro td como argumento, como lo podemos ver seguidamente.



Como podemos observar en la variable tr nos muestra los siguientes datos. Object { row: 0, column: 2, columnVisible: 2 } column: 2 columnVisible: 2 row: 1

A medida que hacemos click en una fila determinada el row va ir variando dependiendo en qué fila se encuentra ubicada, en este caso hicimos click en el primer elemento, que corresponde a la fila número 0, si hacemos click en el tercer elemento nos devolverá el row en 2, por consiguiente aquí tenemos lo necesario para extraer los datos de nuestra tabla. Volvamos a nuestro archivo funciones.js y modifiquemoslo.

```
$('#table_id tbody').on('click', 'button[rel="edit"]', 'tr', function(){
    var td = $(this).closest('td, li');
    var tr = table.cell(td).index()
    var data = table.row(tr.row).data()
    console.log(data);
})
```



Obtuvimos los datos de nuestra tabla de la fila que deseamos modificar de una forma diferente, en este caso, esta es la manera más efectiva para obtener los datos ya que no tendremos problemas al reducir el ancho de nuestra página.

Creando la ventana modals con los datos recuperados

Una vez obtenido los datos de nuestra fila y guardado en la variable data, nos queda abrir el modal para poder agregar dentro del mismo el formulario con los datos provenientes del mismo y volver a enviarlo a nuestro back-end para que lo procese, valide y lo guarde en su caso, para ello en nuestro archivo html necesitamos crear un nuevo input dentro del modals con el atributo hidden para que se oculte, esto en razón de que necesitamos enviar junto con el formulario el id perteneciente a la categoría la cual queremos editar, ya que en nuestro archivo views.py necesitamos instanciar el objeto a fin de pasarlo a nuestro CategoryForm junto con el registro modificado para que éste lo valide.

Primeramente vayamos a nuestro archivo listCategory.html que quedará de la siguiente manera.

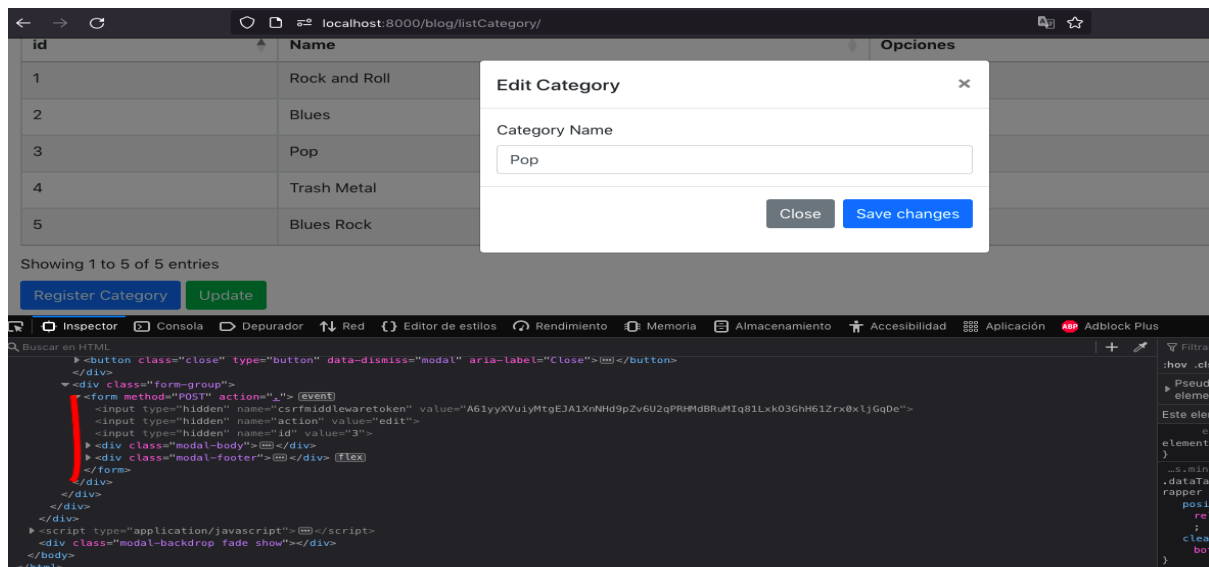
```
<div class="form-group">
  <form method="POST" action=".">
    {% csrf_token %}
    <input type="hidden" name="action" value="">
    <input type="hidden" name="id" value="">
    <div class="modal-body">
      <label>Category Name</label>
      {{form.name}}
    </div>
    <div class="modal-footer">
      <button type="button" class="btn btn-secondary" data-dismiss="modal">Close</button>
      <button type="submit" class="btn btn-primary">Save changes</button>
    </div>
  </form>
</div>
```

Vemos como hemos agregado un input con el name igual a 'id', con un valor que es igual a una cadena vacía, esto para agregar posteriormente el valor del id de nuestra categoría que hayamos de seleccionar con la opción del botón edit. Arriba del mismo se encuentra el input con un valor vacío, que como ya sabemos le pasaremos una acción para que podamos enviarlo a nuestra vista y que este a través de condicionales realice la acción que corresponde en este caso el de editar un registro.

En nuestro archivo funciones.js en el bloque de código que creamos con el fin de obtener los datos de la fila seleccionada con el botón edit, agreguemos las siguientes líneas de código.

```
$('#table_id tbody').on('click', 'button[rel="edit"]', 'tr', function(){
  var td = $(this).closest('td, li');
  var tr = table.cell(td).index()
  var data = table.row(tr.row).data()
  $('#modal_category').modal('show');
  $('#modal_category form')[0].reset();
  $('#modal_title').text('Edit Category');
  $('input[name="action"]').val('edit');
  $('#id_name').val(data[1]);
  $('input[name="id"]').val(data[0]);
});
```

Aquí prácticamente utilizamos el código ya escrito anteriormente en la parte donde llamamos al modals haciendo click el botón Register Category, con algunas modificaciones, le hemos agregado al atributo action del input el valor de edit, al input name le hemos asignado el valor data[1] que corresponde a la categoría seleccionada y obtenida previamente, así también hemos puesto un título 'Edit Category'.



Como se puede observar en la imagen precedente, tenemos el nombre de la categoría que queremos editar dentro del input como así también todos los demás datos mencionados anteriormente, ahora debemos modificar el registro y hacer un submit para enviarlo a nuestra vista para que lo procese, valide y lo guarde en su caso.

Procesando los datos en el back-end

Hasta el momento, hemos obtenido los datos de nuestro registro que necesitamos modificar, lo hemos mostrado en la ventana modals, hemos agregado el id y la acción que queremos realizar, ahora nos queda enviarlo a nuestra vista.

archivo views.py

```
def listCategory(request):
    data = {}
    if request.method == 'GET':
        template_name = 'listCategory.html'
        form = CategoryForm()

        return render(
            request,
            template_name,
            {
                'form': form,
            }
        )

    if request.method == "POST":
        try:
            action = request.POST['action']
            if action == 'search':
                data = list(Category.objects.all().values_list())
                return JsonResponse(data, safe=False)
            elif action == 'create':
                form = CategoryForm(request.POST)
                if form.is_valid():
                    form.save()
                    return JsonResponse(data, safe=False)
            else:
                data['error'] = f"Category {request.POST['name']} already exists"
```

```
        return JsonResponse(data, safe=False)
    elif action == 'edit':
        pk = request.POST['id']
        objeto = Category.objects.get(pk=pk)
        form = CategoryForm(request.POST, instance=objeto)
        if form.is_valid():
            form.save()
            return JsonResponse(data, safe=False)
        else:
            data['error'] = f"Category {request.POST['name']} already exists"
            return JsonResponse(data, safe=False)
    except Exception as e:
        data['error'] = str(e)
        return JsonResponse(data, safe=False)
```

Lo que hacemos aquí es comparar si el valor del input action es igual a 'edit', para que luego podamos extraer el objeto que queremos modificar con el id enviado con nuestro formulario dentro de una variable objeto, para luego pasarlo como parámetro de una instancia a nuestro CategoryForm junto el request.POST, luego verificamos si el formulario enviado es válido y en caso afirmativo lo guardamos y en caso contrario enviamos un mensaje de error tal como lo hicimos con el create category. Nuestra vista luce de la siguiente manera.

En el caso de no instanciar nuestro registro y no pasarlo a la CategoryForm(request.POST, instance=objeto), lo que hará es crear un nuevo registro en el caso que hemos modificado el nombre de la categoría en nuestra interfaz y nos arrojaría un error de f"Category {request.POST['name']} already exists" es decir que ya existe la categoría, en el caso de ingresar una categoría existente en nuestra base de datos.

Si instanciamos correctamente tal como lo hemos hecho, al modificar un registro en nuestra interfaz y enviarlo a nuestra vista, este lo guardará modificando el registro en la base de datos sin alterar el id o crear un id nuevo. -

ELIMINANDO UN REGISTRO

Únicamente nos resta eliminar un registro de nuestra base de datos desde nuestra interfaz, para ello ya hemos creado previamente el botón delete el cual lo renderizamos a nuestro DataTables.

Capturando el evento click del botón delete renderizado y recuperando los datos del registro seleccionado

Primeramente necesitamos controlar el evento click del botón delete tal como lo hemos realizado con el edit, para ello necesitamos llamar a una función dentro de nuestro archivo funciones.js, el cual se encargará de recuperar los datos de nuestra tabla y mostrarnos una ventana emergente quien nos pondrá una advertencia si queremos eliminar o no el registro, en caso de confirmar enviar los datos del registro a nuestro back-end para que lo procese y lo elimine en su caso. Para la ventana emergente utilizaremos un complemento muy fácil de usar el cual es SweetAlert 2 que lo podemos descargar de la siguiente pagina <https://sweetalert2.github.io/>, este complemento está muy bien documentado con ejemplos claros y sencillos.

Archivo funciones.js

```
$('#table_id tbody').on('click', 'button[rel="delete"]', function(){
    var td = $(this).closest('td, li');
    var tr = table.cell(td).index();
    var data = table.row(tr.row).data();
    var data_id = data[0];
    Swal.fire({
        title: 'Are you sure?',
        text: "You won't be able to revert this!",
        icon: 'warning',
        showCancelButton: true,
        confirmButtonColor: '#3085d6',
        cancelButtonColor: '#d33',
        confirmButtonText: 'Yes, delete it!'
    }).then((result) => {
        if (result.value == true) {
            deleteCategory(data_id);
        }
    })
})
```

Archivo funciones.js

```
function deleteCategory(data){
    var csrftoken = document.getElementsByName('csrfmiddlewaretoken')[0].value
    $.ajax(
        {
            headers: {'X-CSRFToken': csrftoken},
            url : window.location.pathname,
            type: "POST",
            data : {
                id: data,
                action: 'delete'
            },
        },
    )
    .done(function(data) {
        if(!data['error']){
            $('#modal_category').modal('hide');
            Swal.fire({
                position: 'top-end',
                icon: 'success',
                title: 'Se ha eliminado correctamente',
                showConfirmButton: false,
                timer: 1500
            })
            $('#table_id').DataTable().ajax.reload();
            return false
        }else{
            alert(data.error)
        }
    })
    .fail(function(data) {
        alert( "error" );
    })
    .always(function(data) {
    });
}
```

Aquí, al igual que en el editar registro hemos tomado el evento click pero en este caso el de eliminar, llamamos a una función que obtiene los datos de la fila y lo guarda en una variable, como en este caso, únicamente necesitamos enviarle el id de la categoría el cual es único y con eso ya nos bastaría para eliminar dicho registro de la base de datos, seguidamente es abierto automáticamente la ventana emergente de SweetAlert con un mensaje de alerta, con dos botones para confirmar o cancelar la eliminación de nuestro registro, en el caso que hagamos click en confirmar con `.then((result) => toma el resultado de nuestra decisión`, luego con una condicional verificamos si `result.value` es igual a `true` para luego llamar a una función `"deleteCategory(data_id)"` que se encargará de enviar a través de una petición ajax el `data_id` de nuestro registro el cual lo pasamos como argumento a nuestra función `"deleteCategory"`.

Como se pueden percatar la función Ajax es el mismo que el que utilizamos para registrar una nueva categoría, únicamente lo hemos modificado en los datos enviados ya que necesitamos enviarle el id que lo hemos pasado como parámetro y el action que es igual a `'delete'`, el cual nos servirá dentro de nuestro vista para diferenciar el tipo de acción que necesitamos realizar.

Archivo views.py

```
def listCategory(request):
    data = {}
    if request.method == 'GET':
        template_name = 'listCategory.html'
        form = CategoryForm()

    return render(
        request,
        template_name,
        {
            'form': form,
            'cat': Category.objects.all(),
        })

    if request.method == "POST":
        try:
            action = request.POST['action']
            if action == 'search':
                data = list(Category.objects.all().values_list())
                return JsonResponse(data, safe=False)
            elif action == 'create':
                form = CategoryForm(request.POST)
                if form.is_valid():
                    form.save()
                    return JsonResponse(data, safe=False)
                else:
                    data['error'] = f"Category {request.POST['name']} already exists"
                    return JsonResponse(data, safe=False)
            elif action == 'edit':
                pk = request.POST['id']
                obj = Category.objects.get(pk=pk)
                form = CategoryForm(request.POST, instance=obj)
                if form.is_valid():
                    form.save()
                    return JsonResponse(data, safe=False)
                else:
```

```

        data['error'] = f"Category {request.POST['name']} already exists"
        return JsonResponse(data, safe=False)
    elif action == 'delete':
        pk = request.POST['id']
        try:
            obj = Category.objects.get(pk=pk)
            obj.delete()
            return JsonResponse(data, safe=False)
        except Exception as e:
            data['error'] = str(e)
    except Exception as e:
        data['error'] = str(e)
    return JsonResponse(data, safe=False)

```

Nuestro vista ha quedado de la siguiente manera, le hemos agregado un bloque de código nuevo el cual comienza con la condicional `action == 'delete'`, como dijimos precedentemente, en la petición ajax hemos enviado dicho dato, en caso de que `action` sea igual a `delete`, el flujo entra a nuestro bloque de código, que primeramente lo que hace es guardar en una variable `pk` (pueden poner cualquier tipo de variable, `id`, `índice`, `identificador` etc) el `id` del registro seleccionado en nuestra interfaz, luego realizamos un `try` a fin de controlar cualquier tipo de error que pueda suceder, obtenemos el registro que deseamos eliminar, y con `delete()` lo eliminamos, luego hacemos un retorno, y en caso de existir algún tipo de error en caso que suceda nos devolverá este como lo hemos hecho con `Create` y `Edit` categoría.

Con esto tenemos la capacidad de eliminar fácilmente cualquier categoría que deseemos.

Seleccionamos el registro que necesitamos eliminar

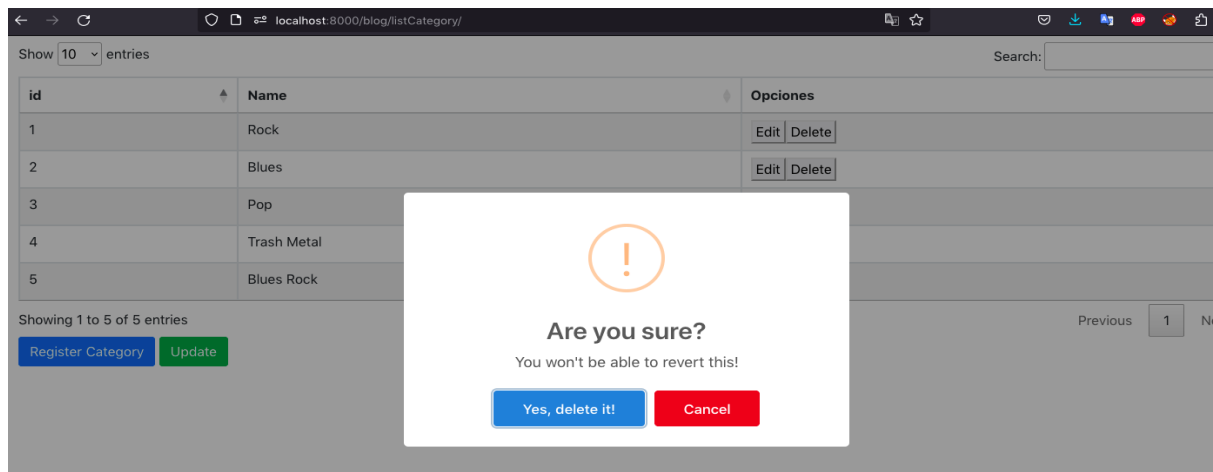
id	Name	Opciones
1	Rock	Edit Delete
2	Blues	Edit Delete
3	Pop	Edit Delete
4	Trash Metal	Edit Delete
5	Blues Rock	Edit Delete

Showing 1 to 5 of 5 entries

Previous 1 Next

[Register Category](#) [Update](#)

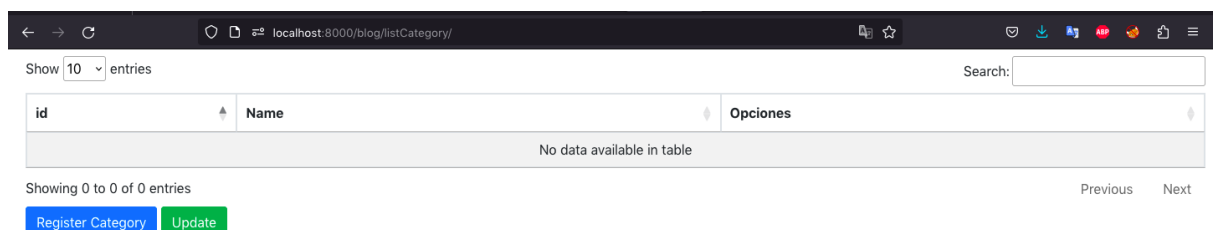
Nos aparecerá una ventana emergente hecha con SweetAlert 2, donde tenemos la opción de cancelar la eliminación o eliminarlo definitivamente.



Si confirmamos, se elimina el registro apareciendo un mensaje donde nos dice que se ha eliminado correctamente, repetimos esta acción hasta el último registro. -



Luego, vemos como nuestra tabla se ha actualizado asincrónicamente, sin recargar toda la página.



Y con esto hemos acabado de terminar nuestra aplicación web CRUD con Django y JavaScript realizando las comunicaciones entre el front-end y el back-end a través de peticiones post con Ajax. -

CONCLUSIÓN

En este trabajo logramos crear una aplicación web CRUD con Django y JavaScript realizando las comunicaciones entre el front-end y el back-end a través de peticiones post con Ajax, desde cero, en el cual utilizamos una base de datos predeterminada de django, en el que creamos una tabla llamada Category, el cual sirve para registrar nombre de categorías musicales, y que cuenta con dos columnas el nombre de la categoría y el id único para cada registro y autoincremental, esto lo hicimos utilizando los models de Django.

Dentro de una plantilla html, creamos una tabla con DataTables en el que mostramos los datos extraídos de nuestra base de datos a través de peticiones asíncronas con Ajax, para luego poder editarlo o eliminarlo en su caso.

FUENTE DOCUMENTALES

- 1 - <https://www.djangoproject.com/>
- 2 - <https://jquery.com/>
- 3 - <https://datatables.net/>
- 4 - <https://sweetalert2.github.io/>
- 5 - <http://api.jquery.com/jquery.ajax/>
- 6 - <https://www.javascript.com/>