

# Introducción a Django Framework

## ¿Qué es Django?

- Django es un framework web creado en 2003, programado en Python, gratuito y libre.
- Fue liberado en 2005 y desde 2008 cuenta con su propia fundación para hacerse cargo de su desarrollo.
- Es utilizado por multitud de grandes empresas como Instagram, Disqus, Pinterest, Bitbucket, Udemy y la NASA
- Promueve el desarrollo ágil y extensible basado en un sistema de componentes reutilizables llamados "apps".
- Algunas de sus mejores características son su mapeador ORM para manejar la base de datos como conjuntos de objetos junto a un panel de administrador autogenerado, así como un potente sistema de plantillas extensibles.
- Funciona sobre distintos tipos de base de datos SQL y es ideal para manejar autenticación de usuarios, sesiones, operaciones con ficheros, mensajería y plataformas de pago.
- No es la mejor alternativa para manejar microservicios ni tampoco servicios que requieran ejecutar múltiples procesos, como las aplicaciones de big data y plataformas con sockets en tiempo real.

## Instalación con Pipenv

- Creamos un directorio **cursodjango** para el proyecto.
- Lo abriremos en VSC para que se inicie la configuración.
- Abriremos la terminal e instalaremos **pipenv**:

```
pip install pipenv
```

- Creamos el entorno virtual instalando django directamente dentro del directorio del proyecto:

```
pipenv install django
```

## Creando el proyecto

- Utilizaremos este comando para crear el proyecto:

```
pipenv run django-admin startproject tutorial
```

- Probaremos a poner el servidor en marcha:

```
cd tutorial  
pipenv run python manage.py runserver
```

- Ya podremos acceder a la url y ver el proyecto:

`http://127.0.0.1:8000/`

- Crearemos un script en el **Pipfile** para simplificar la puesta en marcha:

```
Pipfile
[scripts]
server = "python manage.py runserver"
```

- Ahora podemos ejecutar el servidor con:

```
pipenv run server
```

## Configuración básica

En esta lección vamos a aprender sobre la configuración del proyecto que se realiza en los ficheros del directorio con el mismo nombre que el proyecto:

- **\_\_init\_\_.py**: Sirve para inicializar un paquete en un directorio y también para ejecutar código durante su importación. Desde Python 3.3 su uso no es obligatorio porque los paquetes se crean implícitamente.
- **settings.py**: Es el fichero principal de la configuración, volveremos a él después de ver los demás ficheros.
- **urls.py**: Aquí es donde se configuran las direcciones que escucharán peticiones en las URL del navegador para devolver las distintas páginas de la aplicación web.
- **wsgi.py**: Este último fichero contiene la configuración de la interfaz WSGI, una interfaz para realizar el despliegue del servidor en entornos de producción.

Bien, de vuelta al **settings.py** repasemos las variables de configuración. Si no os interesa profundizar podéis pasar de largo pero conocerlas os ayudará a entender cómo funciona Django:

- **BASE\_DIR**: Esta variable genera la ruta al directorio del proyecto y se utiliza para generar rutas a otros ficheros del proyecto.
- **SECRET\_KEY**: La clave secreta es una variable aleatoria y única generada automáticamente por Django que se utiliza principalmente para tareas criptográficas y generación de tokens.
- **DEBUG**: Cuando esta opción está activada se mostrará información de los errores como respuesta a las peticiones que fallan. Debido a que esta información es privada hay que desactivar el modo debug al publicar el proyecto en Internet.
- **ALLOWED\_HOSTS**: Esta lista contiene las IP y dominios donde queremos permitir que Django se ejecute. Por ejemplo nuestro host local sería el '127.0.0.1' que se encuentra añadido por defecto.

- **INSTALLED\_APPS:** Aquí tenemos otra lista importante que contiene las apps activadas en el proyecto. Las apps son como submódulos que controlan diferentes aspectos del proyecto y por defecto Django trae varias activadas para controlar funcionalidades como el panel de administrador, la autenticación de usuarios y las sesiones. Cuando nosotros hayamos creado nuestras propias apps o queramos utilizar apps externas creadas por la comunidad también tendremos que añadirlas a esta lista.
- **MIDDLEWARE:** Los middlewares son funciones de código que se añaden durante el procesamiento de las peticiones y respuestas a las URL. Django trae muchos por defecto aunque no todos son necesarios.
- **ROOT\_URLCONF:** Esta variable contiene la ruta donde se encuentra el fichero con las URL, por defecto **tutorial.urls** haciendo referencia al directorio **tutorial** y el fichero **urls**.
- **TEMPLATES:** Contiene una configuración completa para manejar los templates o plantillas de Django. Los templates son los ficheros HTML del proyecto y aquí se configura la forma de buscarlos, cargarlos y procesarlos.
- **WSGI\_APPLICATION:** Otra variable que contiene la ruta a la variable **application** del fichero **wsgi.py** para usarla durante el despliegue.
- **DATABASES:** Aquí se define la configuración de acceso a la base de datos. Por defecto se utiliza **SQLite3** y no es necesario configurar nada porque se maneja todo en un fichero. SQLite está bien para el desarrollo y en proyectos pequeños pero en producción es recomendable usar una base de datos centralizada. Django tiene soporte para varias: PostgreSQL, MySQL, MariaDB y Oracle Database.
- **AUTH\_PASSWORD\_VALIDATORS:** Aquí se configuran distintas validaciones para las contraseñas de los usuarios, cosas como la longitud mínima, contraseñas demasiado comunes o numéricas y por tanto inseguras, etc.
- **LANGUAGE\_CODE:** Una de las diferentes opciones de internacionalización de Django que sirven para configurar el idioma, la zona horaria del servidor para manejar fechas y horas, etc. Vamos a poner 'es' para que Django funcione en español.
- **STATIC\_URL:** Finalmente una variable para controlar la ruta a la URL donde se irán a buscar los ficheros estáticos como CSS, JavaScript e imágenes.

## Apps en Django

Cada app de Django es independiente, contiene sus propias definiciones, templates y ficheros estáticos.

El caso es que necesitamos crear como mínimo una app para empezar a trabajar y como vamos a estar desarrollando un pequeño **blog** de prueba podríamos crear esta app con ese mismo nombre:

```
pipenv run python manage.py startapp blog
```

Como véis se crea un directorio con el nombre de la app en nuestro proyecto y dentro hay un montón de nuevos ficheros, los iremos descubriendo poco a poco.

Por ahora sólo nos falta activar la app en el **settings.py** para poder utilizarla:

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

## Modelos en Django

En esta lección introduciremos los modelos de Django, unas clases donde se define la estructura de los datos para almacenarlos en la base de datos.

Nosotros vamos a crear un **blog** muy sencillo así que tendremos que almacenar las entradas. Estas entradas estarán formadas por varios campos, por ejemplo un título y un contenido:

```
blog/models.py  
from django.db import models  
  
class Post(models.Model):  
    title = models.CharField(max_length=200)  
    content = models.TextField()
```

Una vez tenemos el modelo, que es como la estructura de una tabla en la base de datos, tendremos que crear una migración con un registro de los cambios:

```
pipenv run python manage.py makemigrations
```

Una vez hecha la migración tenemos que aplicar los cambios en la base de datos:

```
pipenv run python manage.py migrate
```

Al ser la primera vez que migramos y al tener activadas las apps genéricas del admin, de autenticación y de sesiones se van a crear un montón de campos en la base de datos.

La base de datos la encontraremos en la raíz del proyecto con el nombre **db.sqlite3**. Este fichero se puede consultar y editar con un programa como [DB Browser for SQLite](#). Dentro encontraremos todas las tablas de la base de datos y sus registros.

Sea como sea ya tenemos el modelo **Post** listo para empezar a añadir registro

## Administrador en Django

Existen varias formas de manejar los registros de la base de datos:

- A través del código al responder una petición.
- A través del panel de administrador autogenerado.
- A través de la shell de Django, un intérprete de comandos.

En este pequeño curso no veremos la primera forma, nos limitaremos a utilizar el administrador y experimentar un poco con la shell.

Así que veamos como utilizar el panel de administrador de Django.

Para acceder simplemente tendremos que acceder a la URL **/admin**. Esta dirección no es casual, está definida así dentro del fichero **urls.py** de nuestro proyecto.

```
http://127.0.0.1:8000/admin/login/?next=/admin/
```

Nos pedirá identificarnos un usuario y una contraseña. Este usuario no puede ser un cualquier, debe ser un usuario con permisos para acceder al administrador y como no tenemos ninguno vamos a tener que crearlo.

Para crear nuestro primer superusuario lo haremos desde la terminal:

```
pipenv run python manage.py createsuperuser
```

Una vez creado y con el servidor en marcha podremos identificarnos a acceder al administrador, donde encontraremos una sección llamada **Autenticación y autorización**. Esta sección corresponde a la app **auth** de Django y contiene dos modelos, uno para manejar grupos de permisos y otro con los propios usuarios.

Como véis sin hacer nada ya contamos con un panel donde podemos manejar usuarios cómodamente y añadirles permisos para manejar otras apps. Sin embargo nuestra app **blog** todavía no aparece, eso es porque tenemos que activar los modelos que queramos manejar en el administrador.

Así que vamos a configurar el admin para el modelo **Post** y lo haremos en el fichero **admin.py** de la app **blog**:

```
blog/admin.py
from django.contrib import admin
from .models import Post
```

```
admin.site.register(Post)
```

Simplemente con este cambio ya nos aparecerá nuestra app y podremos empezar a crear, editar y borrar nuevos **Post**.

Con esto ya lo tenemos pero antes quiero enseñaros algunas opciones para personalizar los campos que se muestran en los formularios del modelo.

```
blog/models.py
class Post(models.Model):
    title = models.CharField(max_length=200, verbose_name="Título")
    content = models.TextField(verbose_name="Contenido")
```

Con eso podemos mostrar un nombre diferente en los campos y si quisiéramos mostrar un nombre de modelo diferente lo haremos así:

```
class Meta:
    verbose_name = "entrada"
    verbose_name_plural = "entradas"
```

Por último para listar las entradas mostrando su nombre y no esa referencia rara al objeto que aparece, podemos hacerlo sobrescribiendo el método string del modelo:

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

Óbviamente hay mil cosas más que se pueden configurar, pero hablaremos de eso en futuros cursos.

Esto es más que suficiente por ahora, cread un par más de entradas y seguimos.

## Shell en Django

La shell de Django es un intérprete de comandos que nos permite ejecutar código directamente en el backend, algo muy útil para hacer pruebas y consultas.

```
pipenv run python manage.py shell
```

Una vez dentro la podremos escribir las instrucciones que queramos siempre que sigamos una lógica.

Por ejemplo, si queremos consultar las entradas de la base de datos antes tendremos que importar el modelo **Post** y luego utilizar la sintaxis para hacer la consulta:

```
from blog.models import Post
Post.objects.all()
```

Esta instrucción **objects.all()** nos permite recuperar todos los registros que hay en la tabla de entradas y los almacena en una lista especial de Django llamada `QuerySet`.

También podemos recuperar el primer y último registro muy fácilmente:

```
Post.objects.first()
Post.objects.last()
```

O una entrada a partir de su identificador, un número automático que maneja Django internamente:

```
Post.objects.get(id=1)
```

Todo lo que estamos haciendo son consultas a la base de datos, pero se encuentran abstraídas gracias a la API de acceso al mapeador ORM que nos proporciona Django, donde cada registro se puede manejar como un objeto.

De hecho vamos a crear una entrada y a manipularla un poco:

```
post = Post.objects.create(
    title="Otra entrada", content="Texto de prueba")
```

Al ejecutar lo anterior tendremos nuestra entrada creada en la base de datos, podemos consultar el administrador, pero también la tendremos guardada en la variable **post**:

```
post
```

Podríamos editarla simplemente estableciendo el título como si fuera un objeto normal y corriente:

```
post.title = "Otro título diferente"
```

Eso sí, tendremos que llamar al método **save()** del objeto para guardar los cambios:

```
post.save()
```

Por último si quisiéramos borrar esta entrada que tenemos en la variable podemos hacer con el método **delete()**:

```
post.delete()
```

Y para salir de la shell simplemente llamaremos la función **quit()** desde la terminal:

```
quit()
```

Como habéis visto hemos estado interactuando con la base de datos a través de objetos y método, una forma muy sencilla y cómoda de manejar nuestros registros.

# Vistas y urls en Django

Todo lo que hemos hecho hasta ahora no ha implicado todavía el manejo de peticiones y respuestas.

Las peticiones son las diferentes URL que los clientes piden ver, por ejemplo / es la portada y **/blog/** podría ser nuestro blog.

Para manejar estas peticiones Django utiliza el siguiente flujo:

- El cliente hace la petición a una dirección definida en el **urls.py**.
- Esa dirección está enlazada a una vista, una función definida en el fichero **views.py** y que contiene la lógica que procesará la petición, como por ejemplo consultar el modelo **Post** para ver qué entradas hay en la base de datos.
- Por último esos datos se renderizarán sobre un template HTML y se enviarán al cliente para que éste vea el resultado de la petición.

Este flujo de trabajo se conoce en Django como patrón **MVT** (Modelo - Vista - Template) y nos permite separar muy bien cada proceso de los demás.

Para ver todo esto en acción vamos a programar la vista de nuestra portada y devolviendo un simple texto plano:

```
blog/views.py
from django.shortcuts import render, HttpResponseRedirect

def home(request):
    return HttpResponseRedirect("Bienvenido a mi blog")
```

Ahora tenemos que enlazar esta vista a una URL para poder hacer la petición:

```
tutorial/urls.py
from django.contrib import admin
from django.urls import path
from blog.views import home

urlpatterns = [
    path('', home),
    path('admin/', admin.site.urls),
]
```

Tan simple como esto y si accedemos a la raíz de nuestro sitio ya deberíamos ver como se devuelve el texto plano que devolvemos.

Sin embargo la gracia es renderizar un template HTML bien estructurado, así que vamos a crear uno para nuestra portada.

Prestad mucha atención, para crear un template dentro de la app **blog** tenemos que crear un directorio **templates** en la app y dentro otro directorio llamado con el mismo nombre que la app, en nuestro caso **blog**.



Se hace de esta forma porque Django carga en memoria todos los directorios **templates** de las apps unificándolos en el mismo sitio.

En cualquier caso dentro ya podremos crear nuestra plantilla HTML para renderizarla:

```
templates/blog/home.html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Blog</title>
</head>
<body>
    <h1>Bienvenidos a mi blog</h1>
</body>
</html>
```

Por último vamos a cambiar la vista para en lugar de devolver el texto plano renderice esta plantilla HTML que hemos creado:

```
blog/views.py
from django.shortcuts import render, HttpResponse

def home(request):
    return render(request, "blog/home.html")
```

Listo, ya hemos completado la parte de la vista y el template. En la siguiente lección incorporaremos una consulta a la base de datos a través del modelo **Post** y devolveremos las entradas al template para renderizarlas.

## Variables de contexto

Para recuperar las entradas tendremos que cargar el modelo y hacer exactamente lo que hicimos cuando experimentamos en la shell:

```
blog/views.py
from django.shortcuts import render, HttpResponse
from .models import Post

def home(request):
    posts = Post.objects.all()
    return render(request, "blog/home.html")
```

Una vez tenemos las entradas recuperadas tendremos que enviarlas al template y eso lo haremos usando un diccionario de contexto:

```
return render(request, "blog/home.html", {'posts': posts})
```

Los datos que enviamos en el diccionario de contexto se pueden recuperar en el template usando template tags, una de las funcionalidades más atractivas de Django:

```

templates/blog/home.html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Blog</title>
</head>
<body>
    <h1>Bienvenidos a mi blog</h1>
    <p>Estas son las entradas más recientes.</p>
    {{ posts }}
</body>
</html>

```

Lo más increíble de todo es que podemos usar unos template tags especiales que permiten ejecutar lógica de programación en los templates, por ejemplo para recorrer todas las entradas que hay almacenadas en la QuerySet **posts**:

```

{% for post in posts %}
<div>
    <h2>{{ post }}</h2>
</div>
{% endfor %}

```

Por defecto se nos muestra el título porque es lo que devolvemos al sobreescribir el método string del modelo, pero lo que tenemos son objetos por lo que podríamos acceder a sus campos específicos:

```

{% for post in posts %}
<div>
    <h2>{{ post.title }}</h2>
    <p>{{ post.content }}</p>
</div>
{% endfor %}

```

Con esto hemos completado el flujo del patrón Modelo - Vista - Template, recuperamos los datos del modelo y los enviamos al template a través de la vista.

Django se basa siempre en esa idea.

## Páginas dinámicas

En esta última lección vamos a introducir el concepto de las páginas dinámicas añadiendo una nueva página para visualizar las entradas individualmente en lugar de mostrar todo su contenido en la lista.

La clave está en pasar un valor en la URL a través del cual podamos recuperar el registro para renderizarlo. ¿Cuál es ese valor? Pues normalmente será el identificador único del registro (su campo **id**) al que también se puede acceder con el nombre **pk** de primary key:

```

blog/views.py

```

```
def post(request):
    post = Post.objects.get(id=?)
    return render(request, "blog/post.html", {'post': post})
```

Hemos creado la vista y cargaremos el template **post.html** enviándole el objeto **post** en el diccionario de contexto, pero nos falta lo más importante, una forma de recuperar el identificador de la entrada.

Esto se maneja en dos partes, primero en la URL definiendo un parámetro:

```
tutorial/urls.py
from django.contrib import admin
from django.urls import path
from blog.views import home, post

urlpatterns = [
    path('', home),
    path('blog/<id>', post),
    path('admin/', admin.site.urls),
]
```

Y luego en la vista creando ese parámetro como si formara parte de la función:

```
blog/views.py
def post(request, id):
    post = Post.objects.get(id=id)
    return render(request, "blog/post.html", {'post': post})
```

Con esto ya lo tenemos, aunque si accedemos nos dará error de template no encontrado porque todavía no lo hemos creado:

```
templates/blog/post.html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>{{post.title}}</title>
</head>
<body>
    <h1>{{post.title}}</h1>
    <p>{{post.content}}</p>
    <a href="/">Volver a la portada</a>
</body>
</html>
```

Ya sólo tenemos que añadir un enlace a los títulos para movernos a las entradas:

```
blog/templates/blog/home.html
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Blog</title>
```

```
</head>
<body>
  <h1>Bienvenidos a mi blog</h1>
  <p>Estas son las entradas más recientes.</p>
  {% for post in posts %}
  <div>
    <h2>{{ post.title }}</h2>
    <a href="/blog/{{post.id}}">Leer más</a>
  </div>
  {% endfor %}
</body>
</html>
```

Nuestra web ahora tendrá tantas páginas como entradas tengamos en el blog pero nosotros solo hemos creado la plantilla base. O lo que es lo mismo, tenemos una web generada dinámicamente a partir de los registros que hay en la base de datos.

## Proyecto API con Django Rest Framework

En este curso haremos uso de algunas apps de Django para implementar una API totalmente funcional utilizando autenticación con Token, muy útil para utilizar en todo tipo de clientes, desde web a aplicaciones móviles, ya que se basa sobre una capa de peticiones con respuestas JSON.

Os enseñaré que no es necesario ser todo un experto para desarrollar la plataforma y en muy poco tiempo tendremos un genial sistema para gestionar películas.

En él los usuarios que se registren e identifiquen podrán marcarlas como favoritas, así como ordenar y buscar en los viewsets gracias a las utilidades de Django Rest Framework.

**Repositorio:** [https://github.com/rubences/Peliculas\\_Django.git](https://github.com/rubences/Peliculas_Django.git)

### Creando nuestro proyecto

Partiremos de la base que ya sabéis crear entornos virtuales con **Pipenv** tal como explico en el primer curso de Django, así que trabajaremos en un directorio llamado **proyecto\_pelis** :

```
cd proyecto_pelis
pipenv install django
```

Creamos el proyecto:

```
pipenv run django-admin startproject api_pelis
```

Ahora instalamos todos los módulos de Django para este tutorial, a poder ser con estas versiones para no tener contratiempos:

```
pipenv install.djangorestframework, markdown, django-filter
```

Activamos la app *rest\_framework* en el **settings.py**.

En este punto os recomiendo crear algunos atajos de comandos en el **Pipfile** para agilizar el desarrollo:

```
Pipfile
[scripts]
dev = "python manage.py runserver 127.0.0.1:8844"
make = "python manage.py makemigrations"
migrate = "python manage.py migrate"
```

Para poner en marcha nuestro proyecto utilizaremos el atajo así:

```
pipenv run dev
```

Si todo va bien deberíamos tener Django en marcha: <http://127.0.0.1:8844/>.

A continuación crearemos la app que manejará la API:

```
pipenv run python manage.py startapp api
```

La activaremos en el **settings.py** y haremos la migración inicial:

```
pipenv run migrate
```

Finalmente crearemos nuestro usuario administrador:

```
pipenv run python manage.py createsuperuser
```

## Modelo de película y serializador

Creemos nuestro modelo *Pelicula* para la API:

```
api/models.py
from django.db import models

class Pelicula(models.Model):
    titulo = models.CharField(max_length=150)
    estreno = models.IntegerField(default=2000)
    imagen = models.URLField(help_text="De imdb mismo")
    resumen = models.TextField(help_text="Descripción corta")

    class Meta:
        ordering = ['titulo']
```

Migramos los cambios:

```
pipenv run make
pipenv run migrate
```

Ahora vamos a configurar un serializador, éste definirá el contenido de las películas tal como las devolverá la API:

```
api/serializers.py
from .models import Pelicula
from rest_framework import serializers

class PeliculaSerializer(serializers.ModelSerializer):
    class Meta:
        model = Pelicula
        # fields = ['id', 'titulo', 'imagen', 'estreno', 'resumen']
        fields = '__all__'
```

## Programando la viewset y las urls

Tenemos el modelo y el serializador, ya sólo nos falta programar el viewset de DRF:

```
api/views.py
from .models import Pelicula
from .serializers import PeliculaSerializer
from rest_framework import viewsets

class PeliculaViewSet(viewsets.ModelViewSet):
    queryset = Pelicula.objects.all()
    serializer_class = PeliculaSerializer
```

Y ahora añadimos a las urls la ruta de la viewset en la API:

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

from api import views
from rest_framework import routers

router = routers.DefaultRouter()

# En el router vamos añadiendo los endpoints a los viewsets
router.register('peliculas', views.PeliculaViewSet)

urlpatterns = [
    path('api/v1/', include(router.urls)),
    path('admin/', admin.site.urls),
]
```

Ahora podemos navegar a la API para manejar las películas.

Lamentablemente por defecto las viewsets tienen permisos públicos, así que cualquiera podría manejar las películas. Para solucionarlo simplemente añadiremos un permiso por defecto en el **settings.py**:

```
api_pelis/settings.py
REST_FRAMEWORK = {
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly',
    ],
}
```

```
}
```

Éste hará nuestras viewsets de sólo lectura incluso para visitantes no autenticados. Sólo los usuarios identificados con suficientes permisos podrán acceder a las acciones de creación, modificación y borrado.

## Instalación de las dependencias

La autenticación con token es un sistema ideal para usar en webs asíncronas.

El token es un identificador único de la sesión de un usuario que sirve como credenciales de acceso a la API. Si el cliente envía este token en sus peticiones (que generaremos cuando se registra), ésta buscará si tiene los permisos necesarios para acceder a las acciones protegidas.

Desarrollar esta funcionalidad es tedioso, pero por suerte en DRF tenemos una serie de apps que nos harán la vida mucho más fácil, sólo tenemos que configurarlas adecuadamente y en pocos minutos tendremos un sistema de autenticación básico funcionando.

Esas apps son *django-rest-auth* para la autenticación y *django-allauth* para las cuentas de usuario:

```
pipenv install django-rest-auth
pipenv install django-allauth
```

Configurarlas conlleva añadir algunas apps y variables al settings y paths a las urls.

Primero la app *django.contrib.sites* que maneja los sites requeridos por nuestras dependencias:

```
api_pelis/settings.py
django.contrib.sites,
```

Luego *rest\_framework.authtoken* para añadir la autenticación con tokens:

```
rest_framework.authtoken,
```

Dos más de *django-rest-auth*:

```
'rest_auth',
'rest_auth.registration',
```

Y otras tres para *django-allauth*:

```
'allauth',
'allauth.account',
'allauth.socialaccount',
```

Justo debajo de las apps pondremos las siguientes variables:

```
SITE_ID = 1
```

```
ACCOUNT_EMAIL_VERIFICATION = 'none'
ACCOUNT_AUTHENTICATION_METHOD = 'email'
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_UNIQUE_EMAIL = True
ACCOUNT_USERNAME_REQUIRED = False
```

En resumen lo que hacemos es añadir la configuración del site actual, simplemente estableciendo un *SITE\_ID*. Luego toda la parte que manejará la cuenta, que configuraremos sin verificación de email porque para esta pequeña API nos lo necesitamos, así como registro con email en lugar de usuario, algo que Django no permite hacer por defecto y la verdad es que es un puntazo.

Ahora añadiremos el backend de *allauth* a la configuración de Django, sin ello no nos funcionará nada.

```
AUTHENTICATION_BACKENDS = (
    "django.contrib.auth.backends.ModelBackend",
    "allauth.account.auth_backends.AuthenticationBackend"
)
```

Respecto a las URL añadiremos la parte de la autenticación y registro con *rest\_auth*:

```
api_pelis/urls.py
path('api/v1/auth/',
    include('rest_auth.urls')),
path('api/v1/auth/registration/',
    include('rest_auth.registration.urls')),
```

Finalmente migramos de nuevo:

```
pipenv run migrate
```

Si lo hemos hecho todo correctamente, sin mucho problema podremos acceder a la url y registrar un usuario de prueba usando la interfaz web de DRF, de manera que justo después del registro veamos el token generado.

Por cierto, no es necesario poner un Username, sólo el Email y dos contraseñas iguales, eso sí, deben ser bastante seguras.

De la misma forma que registramos un usuario podemos conseguir el token haciendo login en la respectiva URL.

## Interactuando usando cURL

La API de Django nos proporciona la interfaz web, pero ¿cómo crearíamos un usuario o haríamos el login desde un cliente?

Para hacer una prueba vamos a usar cURL, una biblioteca que permite hacer peticiones en un montón de protocolos. Normalmente viene instalada en los sistemas operativos, así que sólo tenemos que abrir la terminal para empezar a probar.



Tanto el registro como el login manejan peticiones con métodos POST, podemos crearlas fácilmente.

Para registrar un usuario lo haremos así (es muy importante usar doble comillas al pasar los argumentos):

```
curl -X POST http://127.0.0.1:8844/api/v1/auth/registration/  
-d "password1=TEST1234A&password2=TEST1234A&email=test2@test2.com"
```

Y para hacer login y conseguir el token:

```
curl -X POST http://127.0.0.1:8844/api/v1/auth/login/  
-d "password=TEST1234A&email=test2@test2.com"
```

La idea trabajando con clientes es crear estas peticiones y almacenar el token en el localStorage del navegador para más adelante pasarlos en las cabeceras de las peticiones que requieran autenticación.

## Modelo de película favorita

El sistema de películas favoritas constará de dos vistas: una para marcar o desmarcar una película como favorita y otra que devolverá todas las películas favoritas del usuario.

Obviamente ambas vistas serán de acceso protegido y requerirán pasar el token de autenticación en las cabeceras.

Sin embargo antes de ponernos con las vistas tenemos que crear el nuevo modelo que relacione las películas con los usuarios en forma de favorito:

```
api/models.py  
from django.contrib.auth.models import User  
  
class PeliculaFavorita(models.Model):  
    pelicula = models.ForeignKey(Pelicula, on_delete=models.CASCADE)  
    usuario = models.ForeignKey(User, on_delete=models.CASCADE)
```

Si esta relación entre una película y un usuario existe podremos entender que el usuario la tiene como favorita, y si la desmarca simplemente la borraremos de la base de datos.

Ahora vamos a añadir el serializador del modelo que utilizaremos luego:

```
api/serializers.py  
from .models import Pelicula, PeliculaFavorita  
  
class PeliculaFavoritaSerializer(serializers.ModelSerializer):  
  
    pelicula = PeliculaSerializer()  
  
    class Meta:  
        model = PeliculaFavorita  
        fields = ['pelicula']
```

El punto es que hacemos uso del otro serializador Pelicula dentro de PeliculaFavorita para conseguir que la API devuelva instancias anidadas, ya veréis como es muy útil.

Por ahora lo dejamos así, vamos a migrar antes de continuar con la siguiente lección:

```
pipenv run make
pipenv run migrate
```

## Vista de película favorita

**Nota:** Las peticiones para crear recursos sin un identificador previo son de tipo POST, mientras que las de tipo PUT se usan para crear/reemplazar las que ya existen a partir de un identificador. En otras palabras, no es no se pueda usar PUT, pero al no tener un identificador explícito es mejor utilizar POST tal como lo encontraréis editado en estos apuntes. Si queréis más información [echad un vistazo a este enlace](#).

La vista para manejar una película favorita se basa en definir un método de petición, que en nuestro caso será de tipo PUT.

En él detectaremos una id de película que pasaremos como parámetro para recuperar la película que queremos marcar como favorita.

Una vez la tengamos recuperada crearemos la relación PeliculaFavorita, y si ya existe la borrarémos haciendo lo contrario:

```
api/views.py
from .models import Pelicula, PeliculaFavorita
from .serializers import PeliculaSerializer,
PeliculaFavoritaSerializer

from django.shortcuts import get_object_or_404

from rest_framework import viewsets, views
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated
from rest_framework.response import Response

class MarcarPeliculaFavorita (views.APIView):
    authentication_classes = [TokenAuthentication]
    permission_classes = [IsAuthenticated]

    # POST -> Se usa para crear un recurso sin un identificador
    # PUT -> Se usa para crear/reemplazar un recurso con un
    identificador

    def post(self, request):

        pelicula = get_object_or_404(
            Pelicula, id=self.request.data.get('id', 0)
        )
```

```

favorita, created = PeliculaFavorita.objects.get_or_create(
    pelicula=pelicula, usuario=request.user
)

# Por defecto suponemos que se crea bien
content = {
    'id': pelicula.id,
    'favorita': True
}

# Si no se ha creado es que ya existe, entonces borramos el favorito
if not created:
    favorita.delete()
    content['favorita'] = False

return Response(content)

```

Haciendo uso de una *APIView* genérica configuraremos el sistema de autenticación y de permisos de la vista para que sea accesible sólo a usuarios autenticados vía token. Ya en la respuesta devolveremos una estructura JSON usando la clase *Response* de DRF.

El nuevo path para acceder a la view quedará así:

```

api_pelis/urls.py
path('api/v1/favorita/', views.MarcarPeliculaFavorita.as_view()),

```

Una vez configurada, si accedemos a la URL veremos como indica explícitamente la autenticación con token para tener acceso a ella.

Antes de probar si podemos añadir películas favoritas a través de cURL vamos a crear algunas películas. Normalmente añadiríamos una configuración para el administrador, pero ya que tenemos la interfaz de DRF nos la podemos ahorrar. Sólo tenemos que acceder al administrador con nuestro super usuario y luego crear las películas desde la interfaz de DRF.

Para crear una petición PUT con cURL hay que estructurarla de la siguiente forma, pasando el token del usuario que marcará una película en las cabeceras:

```

curl -X POST http://127.0.0.1:8844/api/v1/favorita/
-H "Authorization: Token d0e501a9164b2eeef7f87b62581fb391385fd262"
-d "id=1"

```

Se supone que si todo es correcto nos devolverá el estado de la película:

```

{"id":1,"favorita":true}

```

¿Cómo podemos saber si realmente se creó? La forma fácil sería añadir a la base de datos. Usando DB Browser for SQLite es muy fácil abrir el fichero de la base de datos y analizar los datos de las tablas.

Como hemos comprobado el registro existe, si volvemos a ejecutar la petición debería borrarla:

```
{"id":1,"favorita":false}
```

## Vista de películas favoritas

Nuestra siguiente tarea es una vista que devuelva todas las películas favoritas del usuario. Por suerte gracias al serializador que creamos antes es muy fácil:

```
api/views.py
class ListarPelículasFavoritas (views.APIView):
    authentication_classes = [TokenAuthentication]
    permission_classes = [IsAuthenticated]

    # GET -> Se usa para hacer lecturas

    def get(self, request):

        películas_favoritas = PeliculaFavorita.objects.filter(
            usuario=request.user)
        serializer = PeliculaFavoritaSerializer(
            películas_favoritas, many=True)

        return Response(serializer.data)
```

Y el path:

```
api_pelis/urls.py
path('api/v1/favoritas/', views.ListarPelículasFavoritas.as_view()),
```

Se trata de una simple acción de tipo GET donde devolvemos la lista de pelis favoritas serializadas. Para probar si funciona haremos lo propio con cURL:

```
curl -X GET http://127.0.0.1:8844/api/v1/favoritas/
-H "Authorization: Token d0e501a9164b2eeef7f87b62581fb391385fd262"
```

Esto debería devolvernos la lista con la información de todas las películas favoritas, gracias a lo que os comenté de los modelos anidados:

```
[
  {
    "pelicula":{
      "titulo":"El Padrino",
      "imagen":"https://m.media-amazon.com/images/...",
      "estreno":1972
    }
  }
]
```

Como veis no ha sido para nada complejo.

## Top de películas favoritas y filtros

Esta podría bien ser la parte más compleja del proyecto, pues tenemos que encontrar una forma de contar el número de favoritos de cada película y devolver la lista de películas ordenadas a partir de ese campo.

Por suerte me sé un truquillo que nos hará la vida mil veces más fácil.

Lo que vamos a hacer es crear un nuevo campo en la película que almacene el número de favoritos. Este campo lo actualizaremos automáticamente al crearse o borrarse una instancia de PeliculaFavorita.

```
api/models.py
favoritos = models.IntegerField(default=0)
```

Luego abajo del todo crearemos la señal que hará toda la magia, actualizando el contador a partir de una consulta inversa de esas que tanto me gustan:

```
api/models.py
from django.db.models.signals import post_save, post_delete

def update_favoritos(sender, instance, **kwargs):
    count = instance.pelicula.peliculafavorita_set.all().count()
    instance.pelicula.favoritos = count
    instance.pelicula.save()

# en el post delete se pasa la copia de la instance que ya no existe
post_save.connect(update_favoritos, sender=PeliculaFavorita)
post_delete.connect(update_favoritos, sender=PeliculaFavorita)
```

Migramos los cambios:

```
pipenv run make
pipenv run migrate
```

Como utilizamos el campo **all** en el serializador se supone que ya nos devolverá automáticamente este nuevo campo.

Y ya lo tenemos, el detalle maestro lo pondremos añadiendo un par de filtros de ordenamiento y búsqueda al ViewSet de películas que DRF maneja automáticamente. Así permitiremos ordenar las películas por número de favoritos y realizar búsquedas a partir del título:

```
api/views.py
from rest_framework import viewsets, views, filters

class PeliculaViewSet(viewsets.ModelViewSet):
    filter_backends = [filters.SearchFilter, filters.OrderingFilter]
    search_fields = ['titulo']
    ordering_fields = ['favoritos']
```

Ahora desde [la interfaz web de DRF](#) podemos filtrar y ordenar las películas.

# Tutoriales sobre Django

## Utilizar Django en entornos virtuales con Pipenv

Python incluye un gestor de paquetes llamado pip, el problema es que no se puede tener instaladas dos versiones distintas del mismo paquete y si realizamos un proyecto que necesita una versión y luego tenemos que actualizar quizá deja de funcionar.

Los entornos virtuales sirven para crear una instalación de Python aislada a la del sistema, permitiéndonos instalar versiones de los paquetes que queramos.

Hasta hace un tiempo se utilizaba una herramienta llamada virtualenv, cuyo propósito es generar esos entornos aislados. El problema de este método es que es poco práctico porque te obliga a crear los entornos uno a uno identificándolos con un nombre. Ese es un problema común con el gestor de paquetes conda, una alternativa a virtualenv que además permite elegir la versión de Python, pero que también requiere otorgar nombres a los entornos.

Pipenv viene a solucionar ese punto, pues permite crear un entorno virtual individual para cada proyecto sin tener que darle un nombre. Simplemente crea el entorno en el directorio del proyecto y permite manejarlo desde ahí cómodamente.

La limitación que tiene este método es que está ligado a las versiones de Python que tengamos instaladas. Es decir, puedes crear entornos con otras versiones de Python diferentes a la que tienes por defecto, pero las necesitas previamente descargadas en la máquina y establecer las rutas durante la creación del entorno, algo que no sucede en conda.

Utilizar pipenv es muy fácil, para instalarlo haremos lo de siempre:

```
pip install pipenv
```

Luego navegamos con la terminal al directorio de nuestro proyecto y ahí creamos el entorno de la siguiente forma:

```
pipenv shell
```

Esto no sólo lo creará, también lo activará. Lo sabremos porque en la parte delantera de la terminal aparecerá el nombre del proyecto entre paréntesis y un código:

```
(proyecto-hJNodmU0)
```

Para salir de él podemos usar el comando **exit**:

```
(proyecto-hJNodmU0) exit
```

Y para volver a activarlo, simplemente situándonos de nuevo en la carpeta del proyecto hacemos de nuevo:

```
pipenv shell
```

Siempre que tengamos la shell activa, el intérprete **python** hará referencia al del entorno virtual, sin embargo no es obligatorio tener la shell activada, podemos seguir interactuando con el entorno haciendo uso de **pipenv**.

Por ejemplo para instalar un paquete la lógica es simple, como si usáramos pip pero con pipenv:

```
pipenv install <paquete>
```

Incluso se puede instalar un fichero requirements.txt:

```
pipenv install -r requirements.txt
```

Y para desinstalar un paquete:

```
pipenv uninstall <paquete>
```

O si queremos ver la lista de paquetes del entorno organizado por dependencias, podemos hacerlo con:

```
pipenv graph
```

Me gusta utilizar pipenv porque me ahorra tiempo y además Visual Studio Code detecta el entorno automáticamente instalado en el directorio del proyecto.

Para utilizar Django en Pipenv simplemente deberíamos instalarlo en el entorno creado previamente en una carpeta que representará nuestro proyecto:

```
pipenv install django
```

Ahora podemos ejecutar comandos dentro del entorno utilizando **pipenv run** y así crear el proyecto:

```
pipenv run django-admin startproject <proyecto>
```

Para crear una app, nos situaremos en la raíz de nuestro proyecto (donde tenemos el **manage.py**) y lo haremos con:

```
pipenv run python manage.py startapp <app>
```

Siguiendo la misma lógica podemos realizar migraciones, crear superusuarios, etc:

```
pipenv run python manage.py makemigrations
pipenv run python manage.py migrate
pipenv run python manage.py createsuperuser
```

Algo genial es que podemos editar el fichero Pipfile y añadir en él nuestros propios scripts, por ejemplo un script para lanzar el servidor y ahorrarnos algunas palabras. Sólo tenemos que añadir un apartado [scripts] de la siguiente forma:

```
Pipfile
[scripts]
```

```
server = "python manage.py runserver"
```

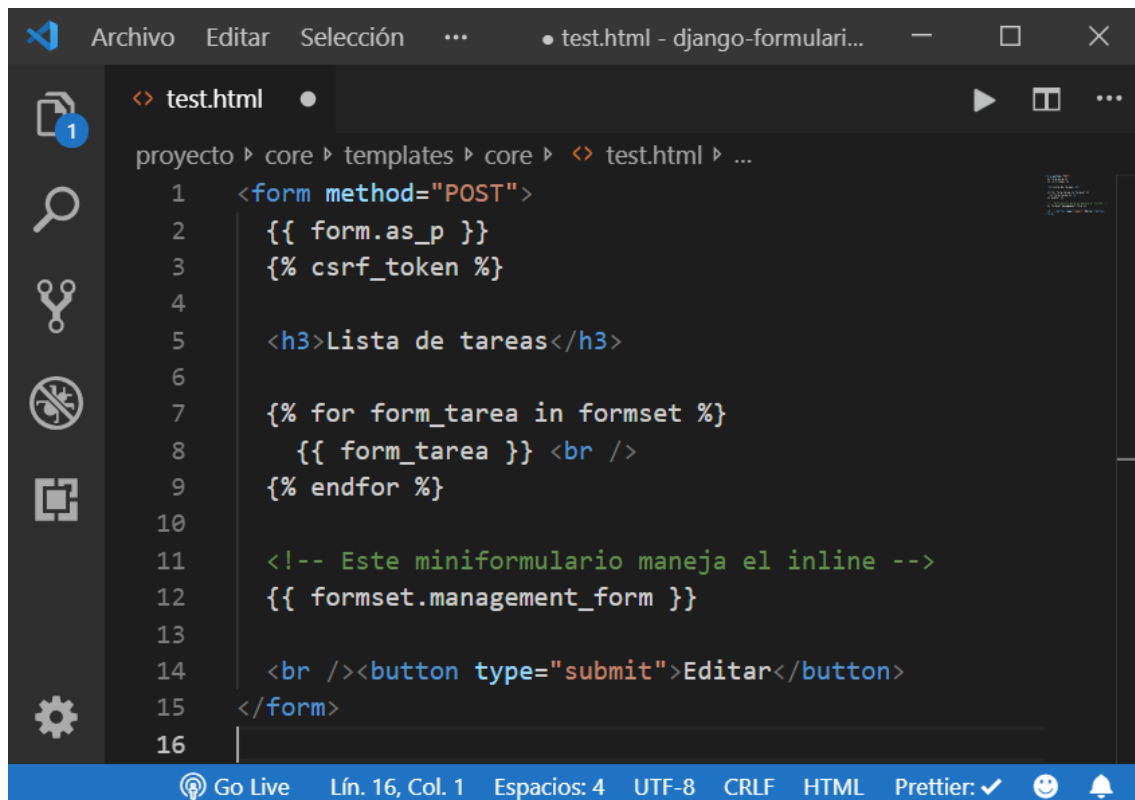
Luego para llamar al script lo haremos con el run y el nombre del script:

```
pipenv run server
```

Todo esto no es más que una muestra del poder de Pipenv, si os parece interesante os dejo la [web oficial](#) y un [tutorial de Real Python](#) que está bastante bien.

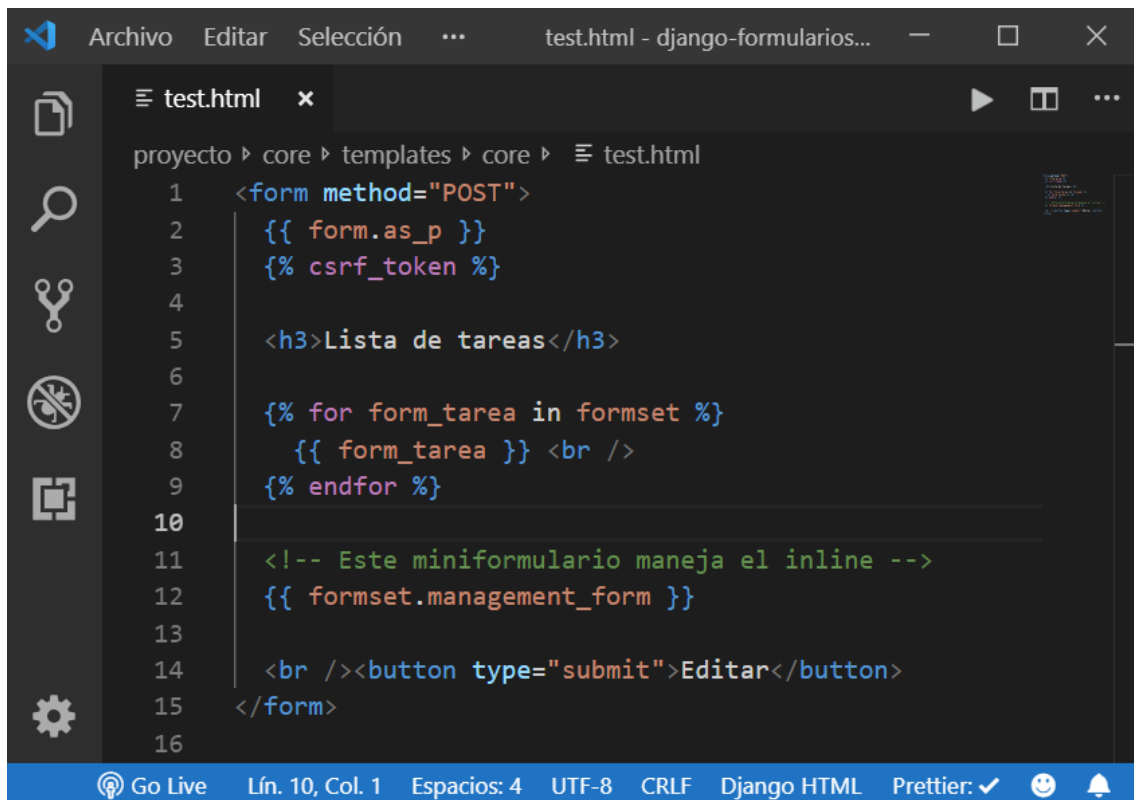
## Colorear sintaxis y autocompletar en Visual Studio Code

Cuando trabajemos con templates HTML de Django en Visual Studio Code veremos que la sintaxis de los template tags no se colorea:



Hay varias extensiones que vienen a solucionar el problema, pero a menudo si arreglan una cosa rompen otra, como por ejemplo el autocompletado de los tags:

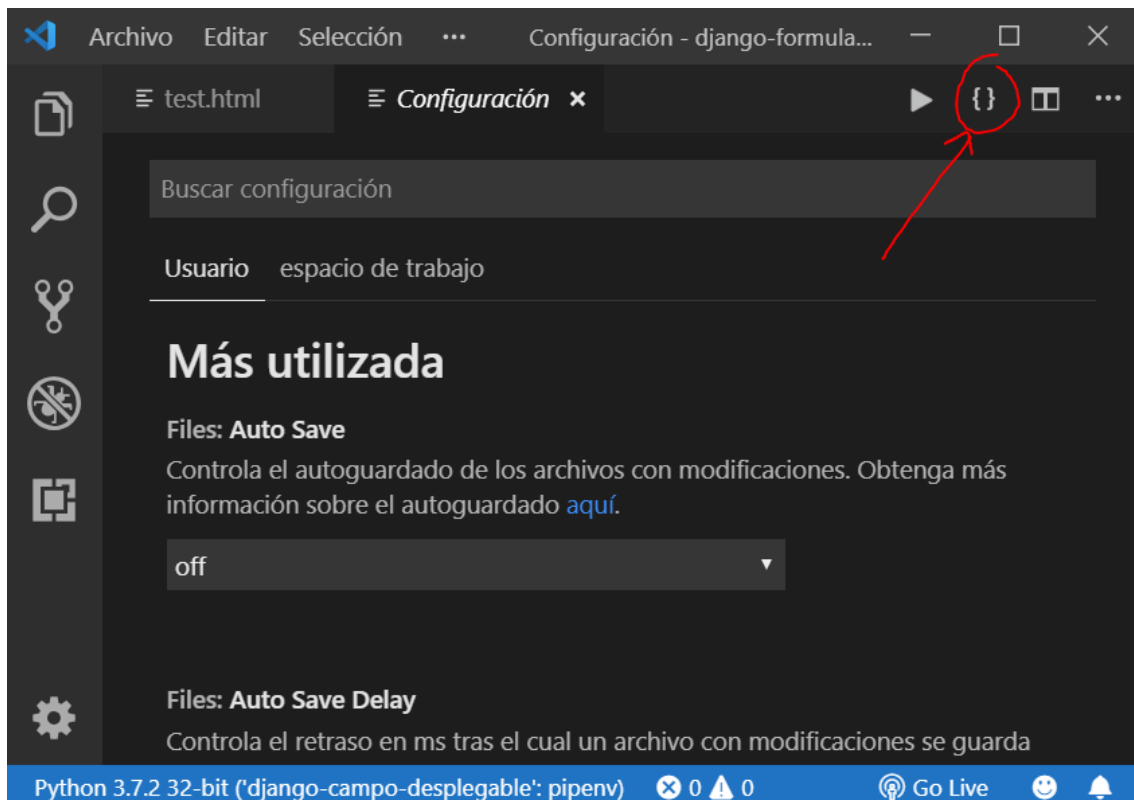




```
1 <form method="POST">
2     {{ form.as_p }}
3     {% csrf_token %}
4
5     <h3>Lista de tareas</h3>
6
7     {% for form_tarea in formset %}
8         {{ form_tarea }} <br />
9     {% endfor %}
10
11     <!-- Este miniformulario maneja el inline -->
12     {{ formset.management_form }}
13
14     <br /><button type="submit">Editar</button>
15 </form>
16
```

La solución es instalar la siguiente extensión llamada Django creada por Baptiste Darthenay, podéis instalarla directamente desde el navegador haciendo clic en el enlace.

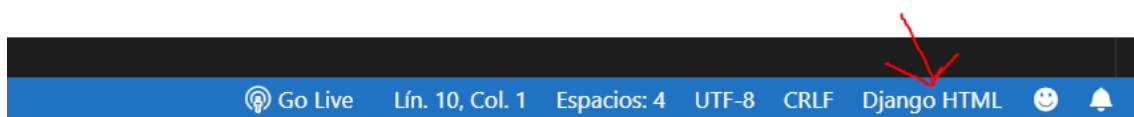
Después de instalarla la clave consiste en configurarla, tendremos que ir a **Archivo > Preferencias > Configuración**, en Windows **Control + ,** y acceder a la configuración JSON en lugar de la interfaz desde el botón superior derecho que luce con dos llaves { }:



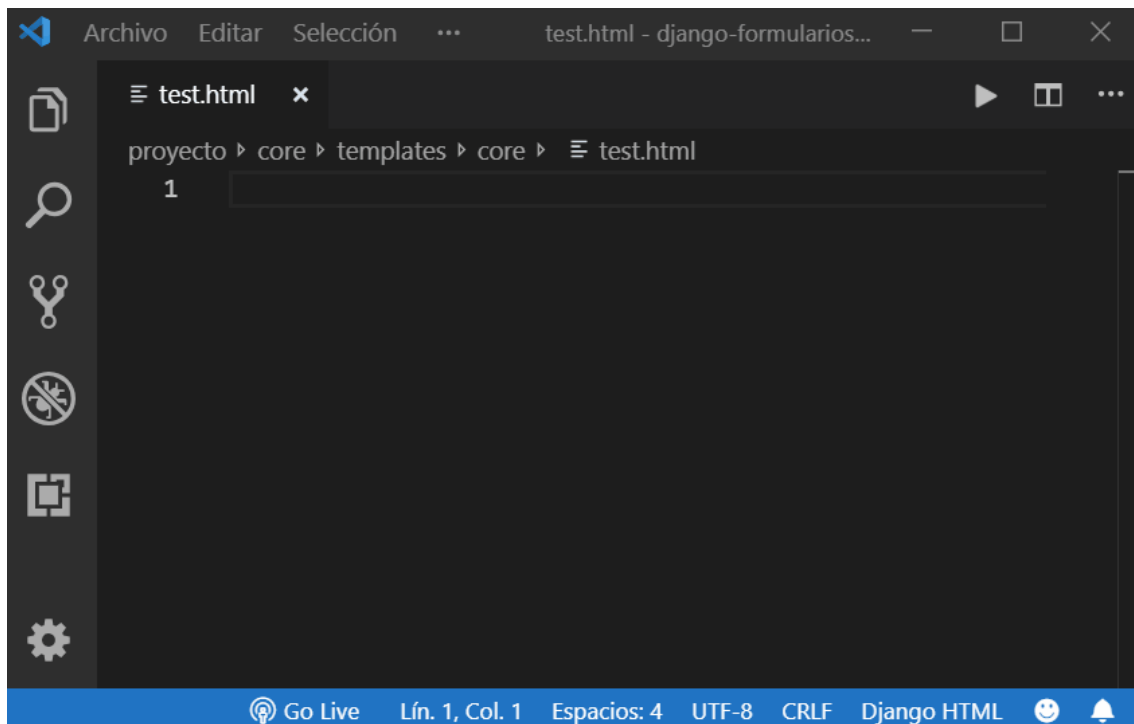
En nuestra configuración añadiremos las siguientes claves JSON con estos valores:

```
settings.json
{
  "files.associations": {
    "**/templates/*.html": "django-html",
    "**/templates/*": "django-txt",
    "**/requirements{/**,*}.{txt,in}": "pip-requirements"
  },
  "emmet.includeLanguages": { "django-html": "html" },
}
```

Una vez lo tengamos reiniciamos Visual Studio Code, abrimos un template de Django y nos aseguramos de tener marcada la opción **Django HTML** en la sintaxis del documento:



Una vez lo tengamos ya deberíamos ser capaces de usar los template tags de Django así como el autocompletado, tanto de HTML como de los template tags usando el tabulador:



## Filtrar un modelo por un campo utilizando un formulario

Con Django podemos crear filtros para nuestros modelos de forma relativamente sencilla jugando con los campos de un formulario y capturándolos en la vista para aplicarlos en las queryset.

Supongamos el siguiente proyecto donde tenemos un modelo **Persona** muy simple:

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

class Persona(models.Model):
    nombre = models.CharField(max_length=100)
    edad = models.SmallIntegerField()
```

Añadimos varias personas a través de nuestro panel de administración:

<input type="checkbox"/>	NOMBRE	EDAD
<input type="checkbox"/>	Alberto	17
<input type="checkbox"/>	Adrián	13
<input type="checkbox"/>	Isabel	43
<input type="checkbox"/>	María	27
<input type="checkbox"/>	Juan	35
5 personas		

Para devolver una lista de nuestras personas utilizaremos una consulta básica al modelo que devuelva todas sus instancias:

```
views.py
from django.shortcuts import render
from .models import *

def home(request):
    personas = Persona.objects.all()

    return render(request, "core/home.html", {'personas': personas})
```

El template que visualizaría el contenido contendría sin mucha complicación un bucle for para recorrer las personas y mostrarlas:

```
home.html
<body>
  <h2>Lista de personas</h2>

  <ul>
    {% for persona in personas %}
      <li>{{persona.nombre}}, {{persona.edad}} años</li>
    {% endfor %}
  </ul>
</body>
```

Este sería el resultado:

## Lista de personas

- Juan, 35 años
- María, 27 años
- Isabel, 43 años
- Adrián, 13 años
- Alberto, 17 años

Ahora viene lo interesante, ¿cómo podemos añadir un filtro para mostrar sólo las personas que tengan una edad mínima?

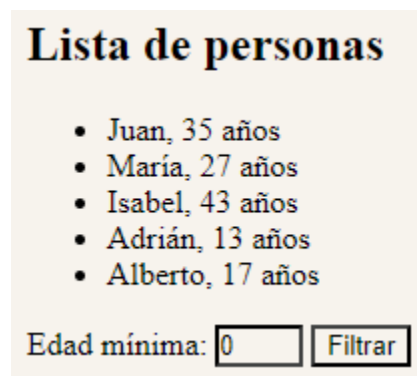
Lo primero sería añadir un formulario que valide contra la propia vista un campo con la edad mínima:

```
home.html
<body>
  <h2>Lista de personas</h2>

  <ul>
    {% for persona in personas %}
      <li>{{persona.nombre}}, {{persona.edad}} años</li>
    {% endfor %}
  </ul>

  <form action="/" method="POST">
    Edad mínima:
    <input type="number" name="edad" value="0" style="width:40px"
/ >
    <input type="submit" value="Filtrar">
    {% csrf_token %}
  </form>
</body>
```

Quedaría así:



The screenshot shows a web page with a title 'Lista de personas'. Below the title is a bulleted list of five people: Juan, 35 años; María, 27 años; Isabel, 43 años; Adrián, 13 años; and Alberto, 17 años. At the bottom of the page, there is a form labeled 'Edad mínima:' followed by a text input field containing the number '0' and a button labeled 'Filtrar'.

Lo único a comentar sería el uso obligatorio del token csrf entre los tags **form** para proteger el formulario de las peticiones entre sitios cruzados.

Para procesar el campo con el **name="edad"**, buscaríamos ese campo en el diccionario **POST** de la petición y lo transformaríamos a número entero para poder utilizarlo en el filtro del queryset:

```
if request.POST.get('edad'):
    edad = int(request.POST.get('edad'))
```

El código final de la vista, una vez aplicado el **filter** quedaría de esta forma:

```
views.py
from django.shortcuts import render
from .models import *

def home(request):
```

```

personas = Persona.objects.all()
edad = 0 # Filtro por defecto
if request.POST.get('edad'):
    edad = int(request.POST.get('edad'))
    personas = personas.filter(edad__gte=edad)
return render(request, "core/home.html", {'personas': personas,
'edad':edad})

```

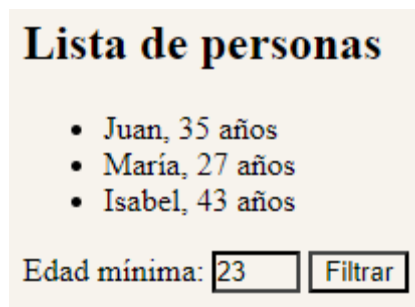
Fijaros como envió el propio campo edad de nuevo al template, así podríamos mostrarlo como valor del input:

```

<input type="number" name="edad" value="{{ edad }}" style="width:40px"
/ >

```

Con esto ya tendríamos nuestro sistema de filtrado para la queryset original con todas las personas aplicándole el filter sólo en caso de recibir el parámetro con la edad por POST:



## Crear, editar y borrar instancias de modelos con formularios

Una de las necesidades más comunes en Django es proveer una interfaz para crear, editar y borrar datos de un modelo.

Django cuenta con un tipo de formularios llamados **ModelForm** que podemos utilizar para gestionar los modelos de una forma cómoda y fácil.

**Nota:** Utilizaremos como base el tutorial anterior de **filtrar un modelo por un campo utilizando un formulario**.

Supongamos que tenemos un modelo de persona y queremos implementar una vista con un formulario para crear nuevas personas:

```

models.py
from django.db import models

class Persona(models.Model):
    nombre = models.CharField(max_length=100)
    edad = models.SmallIntegerField()

```

Lo primero que necesitamos es crear un **Model Form** para manejar este modelo, lo haremos en un fichero **forms.py** dentro de la app:

```
forms.py
from django.forms import ModelForm
from .models import Persona

class PersonaForm(ModelForm):
    class Meta:
        model = Persona
        fields = ['nombre', 'edad']
```

Simplemente tenemos que indicar el modelo del formulario y los campos que vamos a manejar.

Una vez hecho implementaremos una vista para procesar este formulario de creación:

```
views.py
def add(request):
    # Creamos un formulario vacío
    form = PersonaForm()

    # Comprobamos si se ha enviado el formulario
    if request.method == "POST":
        # Añadimos los datos recibidos al formulario
        form = PersonaForm(request.POST)
        # Si el formulario es válido...
        if form.is_valid():
            # Guardamos el formulario pero sin confirmarlo,
            # así conseguiremos una instancia para manejarla
            instancia = form.save(commit=False)
            # Podemos guardarla cuando queramos
            instancia.save()
            # Después de guardar redireccionamos a la lista
            return redirect('/')

    # Si llegamos al final renderizamos el formulario
    return render(request, "core/add.html", {'form': form})
```

Esta vista la llamaremos en una URL específica, por ejemplo `/add/`:

```
proyecto/urls.py
urlpatterns = [
    # ...
    path('add', views.add),
]
```

Finalmente renderizaremos el formulario en el template de la siguiente forma:

```
add.html
<form method="POST">
    {{ form.as_p }}
    {% csrf_token %}
    <button type="submit">Crear</button>
</form>
```

El resultado se vería así:

## Nueva persona

Nombre:

Edad:

---

[Listar personas](#)

Para modificar una instancia el proceso es muy similar al de crearlas, con la peculiaridad de que debemos recuperar la instancia y rellenar el formulario con su información. La vista quedaría así:

```
views.py
def edit(request, persona_id):
    # Recuperamos la instancia de la persona
    instancia = Persona.objects.get(id=persona_id)

    # Creamos el formulario con los datos de la instancia
    form = PersonaForm(instance=instancia)

    # Comprobamos si se ha enviado el formulario
    if request.method == "POST":
        # Actualizamos el formulario con los datos recibidos
        form = PersonaForm(request.POST, instance=instancia)
        # Si el formulario es válido...
        if form.is_valid():
            # Guardamos el formulario pero sin confirmarlo,
            # así conseguiremos una instancia para manejarla
            instancia = form.save(commit=False)
            # Podemos guardarla cuando queramos
            instancia.save()

    # Si llegamos al final renderizamos el formulario
    return render(request, "core/edit.html", {'form': form})
```

La URL definirá un campo numérico donde pasaremos el identificador de la instancia para poder recuperarlo:

```
proyecto/urls.py
urlpatterns = [
    # ...
    path('edit/<int:persona_id>', views.edit),
]
```

Respecto al template, sería el mismo que usamos para crear la instancia, pero podemos utilizar una plantilla diferente para adaptar el texto informativo y mostrar **Editar** en lugar de **Crear** en el botón:

```
edit.html
<form method="POST">
    {{ form.as_p }}
    {% csrf_token %}
    <button type="submit">Editar</button>
```



</form>

Si tenemos una lista de objetos podemos mostrar un enlace para ir al formulario de edición fácilmente creando la URL **/edit/instancia.id**:

```
<ul>
  {% for persona in personas %}
  <li>
    {{ persona.nombre }}, {{ persona.edad }} años
    <a href="/edit/{{ persona.id }}">Editar</a>
  </li>
  {% endfor %}
</ul>
```

Así quedaría el formulario de edición al editar una instancia:

## Editar persona

Nombre:

Edad:

---

[Crear persona](#) | [Listar personas](#)

Si todo está correcto los cambios quedarán establecidos en la instancia al guardarlos, por eso es buena idea añadir un enlace para visualizar la lista de instancias actualizadas:

## Lista de personas

- Juan, 35 años [Editar](#)
- María, 27 años [Editar](#)
- Isabel, 55 años [Editar](#)
- Adrian, 15 años [Editar](#)
- Alberto, 17 años [Editar](#)
- Carmen, 57 años [Editar](#)

Edad mínima:

---

[Crear persona](#)

Por último podemos añadir una opción para borrar instancias a partir de su identificador.

No necesitamos manejar un formulario, simplemente recuperar la instancia en la vista y borrarla:

```
views.py
def delete(request, persona_id):
    # Recuperamos la instancia de la persona y la borramos
    instancia = Persona.objects.get(id=persona_id)
    instancia.delete()

    # Después redireccionamos de nuevo a la lista
    return redirect('/')
```

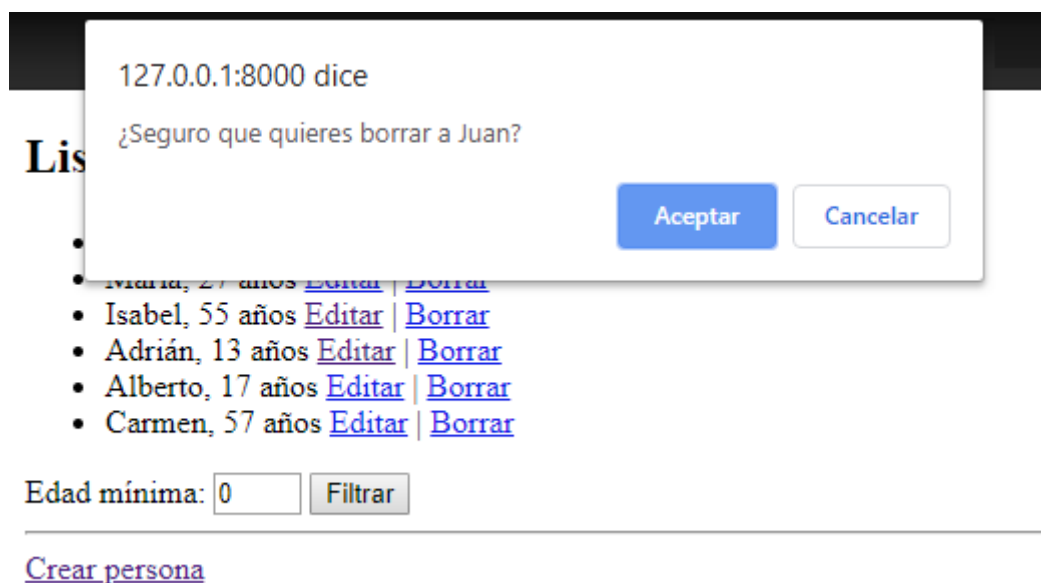
La URL por tanto será casi igual que la de edición:

```
proyecto/urls.py
urlpatterns = [
    # ...
    path('delete/<int:persona_id>', views.delete),
]
```

Sólo necesitamos añadir un enlace en la lista de instancia para borrarlas, recomendablemente con una pequeña confirmación usando JavaScript para prevenir un borrado accidental:

```
<ul>
    {% for persona in personas %}
    <li>
        {{ persona.nombre }}, {{ persona.edad }} años
        <a href="/edit/{{ persona.id }}">Editar</a>
        <a href="/delete/{{ persona.id }}"
            onClick="return confirm('¿Quieres borrar
            {{ persona.nombre }}?');">
            Borrar
        </a>
    </li>
    {% endfor %}
</ul>
```

Al presionar el enlace de borrado nos aparecerá la ventana emergente de confirmación:



Y justo después de confirmar volveremos a la lista donde ya habrá desaparecido la instancia borrada:

## Lista de personas

- María, 27 años [Editar](#) | [Borrar](#)
- Isabel, 55 años [Editar](#) | [Borrar](#)
- Adrián, 13 años [Editar](#) | [Borrar](#)
- Alberto, 17 años [Editar](#) | [Borrar](#)
- Carmen, 57 años [Editar](#) | [Borrar](#)

Edad mínima:

---

[Crear persona](#)

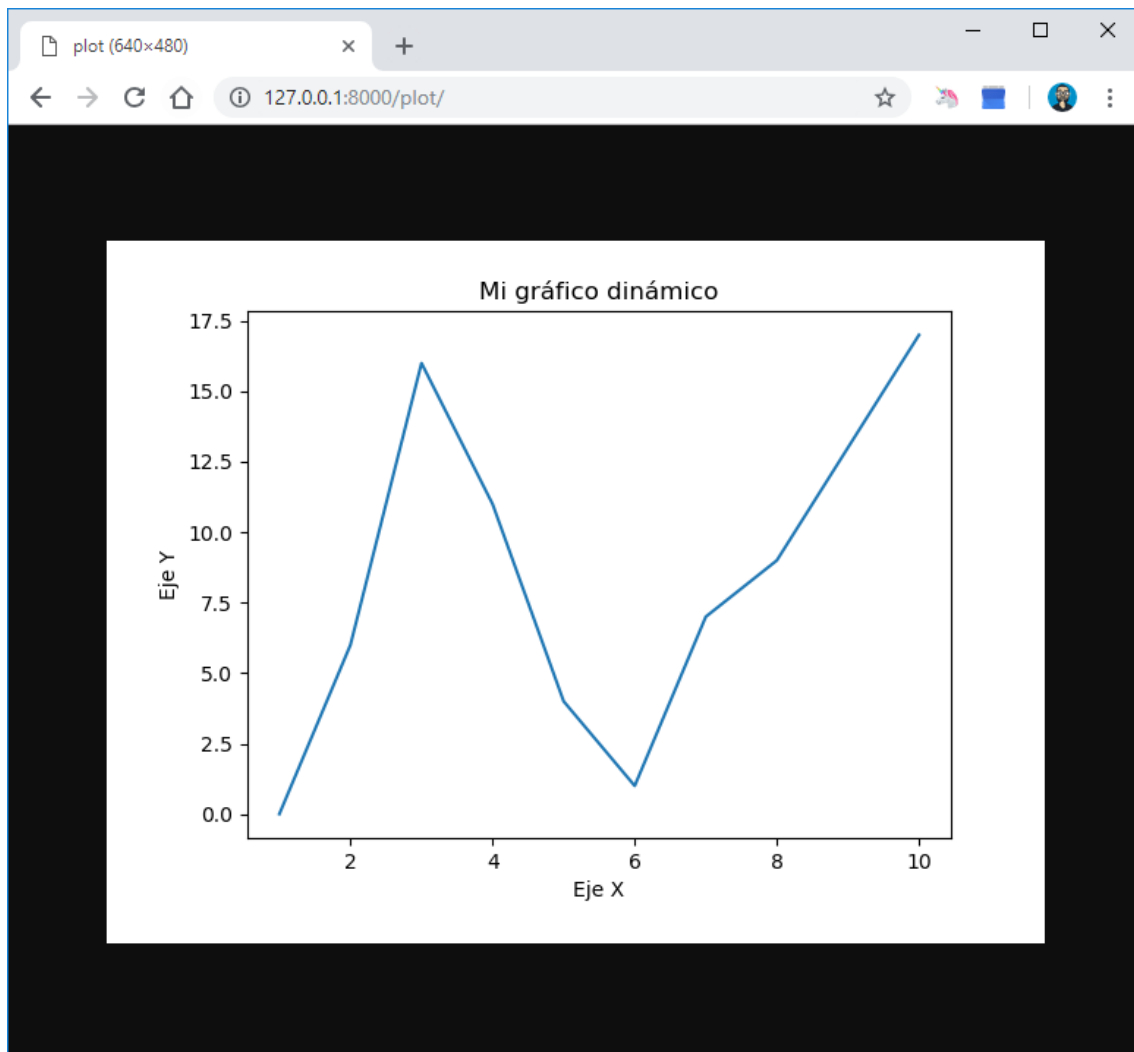
## Mostrar gráficos generados con Matplotlib en Django

Algo que me preguntan muchos mis alumnos de Django es si es posible generar gráficos en Python y mostrarlos en un template.

La respuesta es sí, pero es un poco lioso porque hace falta "renderizar" el canvas de las figuras de Matplotlib sobre un buffer de bytes.

No voy a enseñar nada de Matplotlib porque eso es harina de otro costal, [aquí](#) dejo mis apuntes online, pero sí el proceso para mostrar el gráfico.

La idea de mostrar gráficos con Django se basa en acceder a una url de nuestro proyecto, por ejemplo **/plot/**, y en lugar de renderizar un template, responder con la imagen del gráfico generada en tiempo de ejecución:



Al acceder a la URL que genera el gráfico, ésta devuelve una imagen PNG en lugar de un documento HTML.

El snippet para la vista de Django sería el siguiente:

```
views.py
import io
import matplotlib.pyplot as plt

from django.http import HttpResponse
from django.shortcuts import render
from matplotlib.backends.backend_agg import FigureCanvasAgg

from random import sample

def home(request):
    return render(request, "core/home.html")

def plot(request):
    # Creamos los datos para representar en el gráfico
    x = range(1,11)
    y = sample(range(20), len(x))

    # Creamos una figura y le dibujamos el gráfico
    f = plt.figure()
```

```

# Creamos los ejes
axes = f.add_axes([0.15, 0.15, 0.75, 0.75]) # [left, bottom,
width, height]
axes.plot(x, y)
axes.set_xlabel("Eje X")
axes.set_ylabel("Eje Y")
axes.set_title("Mi gráfico dinámico")

# Como enviaremos la imagen en bytes la guardaremos en un buffer
buf = io.BytesIO()
canvas = FigureCanvasAgg(f)
canvas.print_png(buf)

# Creamos la respuesta enviando los bytes en tipo imagen png
response = HttpResponse(buf.getvalue(), content_type='image/png')

# Limpiamos la figura para liberar memoria
f.clear()

# Añadimos la cabecera de longitud de fichero para más estabilidad
response['Content-Length'] = str(len(response.content))

# Devolvemos la response
return response

```

Esto nos generaría el gráfico que os muestro en la imagen de arriba, que cambiará cada vez que recargamos la página con F5 porque contiene datos aleatorios tomados del módulo **random**.

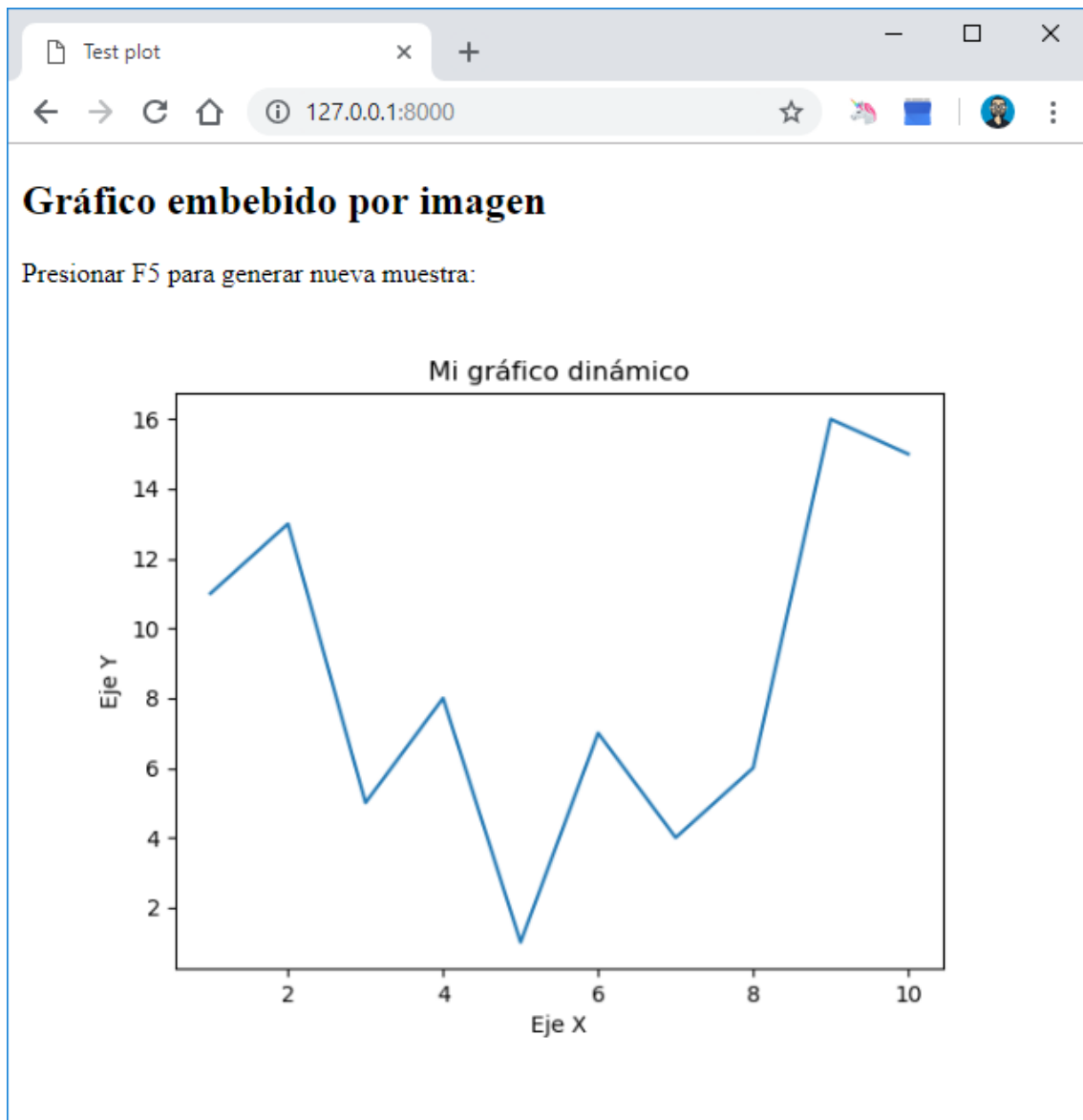
Lo bueno es que podemos insertar esta URL como si fuera una imagen:

```

home.html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Test plot</title>
</head>
<body>
  <h2>Gráfico embebido por imagen</h2>
  <p>Presionar F5 para generar nueva muestra: </p>
  
</body>
</html>

```

Quedando finalmente la página principal de la siguiente forma:



Si a esta idea le sumamos parámetros, ya sea por GET o POST que podemos capturar en la vista, entonces podemos añadir dinamismo y generar los gráficos como nosotros queramos. Por ejemplo pasando un parámetro GET llamado type `/plot/?type=1` donde luego con `request.GET.get['type']` podamos seleccionar el tipo de gráfico o lo que se os ocurra.

## Implementar sistema clásico de registro, login y logout

Este tutorial tiene como objetivo mostrar cómo utilizar las funciones que ofrece Django para registrar y autenticar usuarios utilizando sus apps y formularios internos. No enseñaré a programar funciones extendidas como podrían ser la de cambiar la contraseña o enviar emails de verificación, pues serían funcionalidades para tratar en tutoriales a parte.

Para este experimento vamos a suponer que necesitamos crear una sección privada sólo para usuarios registrados. Esta área exclusiva la manejaremos dentro de una app llamada **users** que también gestionará los formularios de inicio de sesión y login:

```
python manage.py startapp users
```

Una vez la tengáis creada no olvidéis activarla en el **settings.py**.

Tendremos básicamente 4 vistas en la aplicación de usuarios:

- **welcome:** Manejará la bienvenida al área para miembros y redireccionará a la vista de identificación si el usuario no ha iniciado la sesión.
- **register:** Manejará el formulario de registro de usuarios y autenticará al usuario automáticamente al registrarse.
- **login:** Manejará el formulario de identificación de usuarios y redireccionará a la portada si las credenciales son correctas.
- **logout:** Manejará la acción de cerrar la sesión y redirecciona a la vista de la portada de nuevo.

Podemos crearlas inicialmente con el mínimo contenido:

```
views.py
from django.shortcuts import render, redirect

def welcome(request):
    return render(request, "users/welcome.html")

def register(request):
    return render(request, "users/register.html")

def login(request):
    return render(request, "users/login.html")

def logout(request):
    # Redireccionamos a la portada
    return redirect('/')
```

Las URL que las manejarán serán las siguientes:

```
proyecto/urls.py
from django.contrib import admin
from django.urls import path
from users import views

urlpatterns = [
    path('', views.welcome),
    path('register', views.register),
    path('login', views.login),
    path('logout', views.logout),

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

Vamos a empezar con el **logout** porque es la acción más sencilla, sólo tenemos que llamar a la función de mismo nombre que encontraremos en el módulo **django.contrib.auth**. Os sugiero importar la función con otro nombre porque de esa forma podemos usar **logout** en la función de la vista:

```
views.py
from django.contrib.auth import logout as do_logout

# ...

def logout(request):
    # Finalizamos la sesión
    do_logout(request)
    # Redireccionamos a la portada
    return redirect('/')


```

Tan sencillo como esto.

A continuación nos centraremos en añadir una validación a la portada que redirija al usuario al **login** en caso de no estar autenticado, así protegeremos su contenido:

```
views.py
def welcome(request):
    # Si estamos identificados devolvemos la portada
    if request.user.is_authenticated:
        return render(request, "users/welcome.html")
    # En otro caso redireccionamos al login
    return redirect('/login')


```

El contenido de la portada podría ser el siguiente:

```
welcome.html
<h2>Área para miembros</h2>
<p>
    Bienvenido <b>{{request.user.username}}</b>,
    esta página es exclusiva para usuarios registrados.
</p>
<hr />
<a href="/logout">Cerrar sesión</a>


```

Al añadir este código si intentamos acceder a la raíz del sitio / nos redireccionará al **/login** que aún no hemos creado. En caso de ver la portada podría ser por tener una sesión activa previamente desde el panel de administrador, ya que se gestionan con la misma app interna de Django. Si la cerráis desde el enlace inferior os llevará al **login**.

El formulario de identificación es la cosa más sencilla del mundo, sólo necesitamos un campo para el nombre del usuario y otro para la contraseña. Podríamos crearlos manualmente pero también podemos usar los **built-in** forms de Django.

Así que vamos a importar el formulario de autenticación llamado **AuthenticationForm** y dejaremos que él lo gestione todo, nosotros sólo lo validaremos e iniciaremos la sesión si la información es correcta:



```

views.py
from django.contrib.auth import authenticate
from django.contrib.auth.forms import AuthenticationForm
from django.contrib.auth import login as do_login

# ...

def login(request):
    # Creamos el formulario de autenticación vacío
    form = AuthenticationForm()
    if request.method == "POST":
        # Añadimos los datos recibidos al formulario
        form = AuthenticationForm(data=request.POST)
        # Si el formulario es válido...
        if form.is_valid():
            # Recuperamos las credenciales validadas
            username = form.cleaned_data['username']
            password = form.cleaned_data['password']

            # Verificamos las credenciales del usuario
            user = authenticate(username=username, password=password)

            # Si existe un usuario con ese nombre y contraseña
            if user is not None:
                # Hacemos el login manualmente
                do_login(request, user)
                # Y le redireccionamos a la portada
                return redirect('/')

            # Si llegamos al final renderizamos el formulario
            return render(request, "users/login.html", {'form': form})

```

El template quedaría de la siguiente forma, dejando que sea el propio Django quién renderice el formulario:

```

login.html
<h2>Iniciar sesión</h2>
<form method="POST">
    {{ form.as_p }}
    {% csrf_token %}
    <button type="submit">Login</button>
</form>
<hr />
<a href="/register">Registrar usuario</a>

```

Con esto ya tendremos implementada la identificación:

## Iniciar sesión

Nombre de usuario:

Contraseña:

Login

---

[Registrar usuario](#)

Si no tenéis un usuario os recomiendo crear uno desde la terminal para probarlo, pero recordad antes hacer una migración inicial:

```
python manage.py migrate
python manage.py createsuperuser
```

Además lo interesante es que al ser un formulario integrado soporta la traducción dependiendo del idioma que tenemos configurado en Django y también es capaz de detectar los errores e informar si el usuario no es correcto.

## Área para miembros

Bienvenido **admin**, esta página es exclusiva para usuarios registrados.

---

[Cerrar sesión](#)

Y por último vamos con la funcionalidad de añadir nuevos usuarios.

Se maneja de forma muy parecida al **login**, ya que también hay un formulario integrado para manejar esta situación, se trata de **UserCreationForm**:

```
views.py
from django.contrib.auth.forms import UserCreationForm

# ...

def register(request):
    # Creamos el formulario de autenticación vacío
    form = UserCreationForm()
    if request.method == "POST":
        # Añadimos los datos recibidos al formulario
        form = UserCreationForm(data=request.POST)
        # Si el formulario es válido...
        if form.is_valid():
            # Creamos la nueva cuenta de usuario
            user = form.save()
            # Si el usuario se crea correctamente
            if user is not None:
                # Hacemos el login manualmente
                do_login(request, user)
                # Y le redireccionamos a la portada
                return redirect('/')

    # Si llegamos al final renderizamos el formulario
    return render(request, "users/register.html", {'form': form})
```

El template sería prácticamente un calco del de **login**:

```
register.html
<h2>Registrar usuario</h2>
<form method="POST">
    {{ form.as_p }}
    {% csrf_token %}
```

```
<button type="submit">Registrar</button>
</form>
<hr />
<a href="/login">Iniciar sesión</a>
```

Se verá más o menos así:

## Registrar usuario

Nombre de usuario:  Requerido. 150 caracteres como máximo. Únicamente letras, dígitos y @/./+/\_

Contraseña:

- Su contraseña no puede asemejarse tanto a su otra información personal.
- Su contraseña debe contener al menos 8 caracteres.
- Su contraseña no puede ser una clave utilizada comunmente.
- Su contraseña no puede ser completamente numérica.

Contraseña (confirmación):  Para verificar, introduzca la misma contraseña anterior.

---

[Iniciar sesión](#)

Este formulario de registro tiene la peculiaridad de contener mucho texto de ayuda a la hora de crear las cuentas, pero si queremos podemos esconder esa información borrando el atributo **help\_text** de los tres campos del formulario:

```
views.py
# Si queremos borramos los campos de ayuda
form.fields['username'].help_text = None
form.fields['password1'].help_text = None
form.fields['password2'].help_text = None

# Si llegamos al final renderizamos el formulario
return render(request, "users/register.html", {'form': form})
```

Así tendríamos un formulario más limpio, aunque conservaremos los mensajes de error si se introduce un nombre de usuario en uso o si las contraseñas no superan la validación mínima:

## Registrar usuario

Nombre de usuario:

Contraseña:

Contraseña (confirmación):

---

[Iniciar sesión](#)

# Extender el UserCreationForm para registrarse con el email

El sistema de usuarios en Django tiene un problema y es que por defecto utiliza únicamente el usuario y la contraseña. Sin embargo en la actualidad es cada vez más común que los sitios en lugar de un "nick" utilicen el correo electrónico como usuario:

Ya os enseñé a implementar un sistema clásico de registro, login y logout, así que vamos a extender esa lógica para obligar al usuario a registrarse e iniciar sesión utilizando su correo electrónico:

## Registrar usuario

Nombre de usuario:

Contraseña:

Contraseña (confirmación):

---

[Iniciar sesión](#)

La solución más sencilla consiste en engañar al usuario, para que en lugar de registrarse con un nombre utilice su correo electrónico.

Así que vamos a extender el formulario de registro de Django **UserCreationForm** para que en lugar de **Nombre de usuario** muestre el texto **Correo electrónico**. Podemos hacerlo dentro del fichero **forms.py** y llamaremos al nuevo formulario por ejemplo **UCFWithEmail**:

```
forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth.models import User

# Extendemos del original
class UCFWithEmail(UserCreationForm):
    # Ahora el campo username es de tipo email y cambiamos su texto
    username = forms.EmailField(label="Correo electrónico")

    class Meta:
        model = User
        fields = ["username", "password1", "password2"]
```

Ahora en la vista sólo tenemos que cambiar el formulario **UserCreationForm** por nuestro **UCFWithEmail**:

```
# Estas son las líneas que cambian
```

```
from .forms import UCFWithEmail
form = UCFWithEmail()
form = UCFWithEmail(data=request.POST)
```

Y ya lo tendremos:

## Registrar usuario

Nombre de usuario:

Contraseña:

Contraseña (confirmación):

---

[Iniciar sesión](#)

Ahora si quisiéramos cambiar el texto **Nombre de usuario** también en el formulario de login:

## Iniciar sesión

Nombre de usuario:

Contraseña:

---

[Registrar usuario](#)

Podemos hacer lo mismo extendiendo el formulario **AuthenticationForm** en nuestro propio **AFWithEmail**:

```
forms.py
from django import forms
from django.contrib.auth.forms import UserCreationForm,
AuthenticationForm
from django.contrib.auth.models import User

# Extendemos del original
class AFWithEmail(AuthenticationForm):
    # Ahora el campo username es de tipo email y cambiamos su texto
    username = forms.EmailField(label="Correo electrónico")

    class Meta:
        model = User
        fields = ["username", "password"]
```

Para usarlo en la vista, igual que antes, sólo tenemos que cambiar la clase del formulario por la nuestra.

```
# Estas son las líneas que cambian
from .forms import UCFWithEmail, AFWWithEmail
form = AFWWithEmail()
form = AFWWithEmail(data=request.POST)
```

Y listo, ya lo tendremos:

## Iniciar sesión

Correo electrónico:

Contraseña:

Login

---

[Registrar usuario](#)

Faltaría sólo arreglar un detalle durante la validación para que en lugar de **Nombre de usuario** muestre **Correo electrónico**, pero es algo tedioso porque el texto se encuentra dentro de las traducciones de Django, sería más fácil diseñar el formulario manualmente, pero bueno, dejando de banda ese detalle esta forma funciona genial:

## Área para miembros

Bienvenido **hola@hektorprofe.net**, esta página es exclusiva para usuarios registrados.

---

[Cerrar sesión](#)

Vamos a ver cómo utilizar **inline forms**, ya sea a través del administrador o en nuestras propias vistas.

Los **inlines** son formularios que surgen al crear modelos relacionados, normalmente de tipo **Foreign Key**.

El concepto se ve muy fácilmente en un ejemplo, así que vamos a utilizar como base el tutorial de **crear, editar y borrar instancias de modelos con formularios** y lo extenderemos un poco.

Supongamos que tenemos este modelo **Persona**:

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

class Persona(models.Model):
```

```

nombre = models.CharField(max_length=100)
edad = models.SmallIntegerField()

def __str__(self):
    return self.nombre

```

Ahora queremos crear diferentes tareas para cada **Persona**, para ello vamos a crear un modelo relacionado llamado **Tarea**, con un nombre de tarea y una relación la persona que la tendrá asignada:

```

models.py
class Tarea(models.Model):
    nombre = models.CharField(max_length=100)
    persona = models.ForeignKey(Persona, on_delete=models.CASCADE)

    def __str__(self):
        return self.nombre

```

Hasta aquí sin mucha complicación.

Entonces, para añadir al panel de administrador estas tareas en el propio formulario de cada persona para manejar sus tareas cómodamente podemos usar inlines.

Podemos registrar el nuevo admin para **Tarea** en el **admin.py**, pero en lugar de hacerlo como un modelo tradicional, lo haremos como un **inline** y lo asignaremos al admin de **Persona**, fijaros:

```

admin.py
# Creamos el inline para el modelo tarea
class TareaInline(admin.TabularInline):
    model = Tarea
    # Mostramos dos inlines acíos por defecto
    extra = 2

class PersonaAdmin(admin.ModelAdmin):
    list_display = ('nombre', 'edad')
    # Registramos el inline en la persona
    inlines = [TareaInline]

```

Con esto ya lo tenemos, si vamos al administrador veremos la nueva estructura y podremos crear nuevas tareas en la parte inferior, teniendo siempre dos huecos libres para añadir otras:

## Modificar persona

[HISTÓRICO](#)

Nombre:

Edad:

### TAREAS

NOMBRE

¿ELIMINAR?

Comprar leche



Sacar al perro



[+ Agregar Tarea adicional.](#)

A parte de la forma **TabularInline** para mostrar los campos horizontalmente también existe **StackedInline** para hacerlo verticalmente:

```
TareaInline(admin.StackedInline)
```

Así es como se vería en nuestro ejemplo de tareas, no se nota mucho porque sólo tenemos un campo:



## Modificar persona

[HISTÓRICO](#)

Nombre:	<input type="text" value="Carmen"/>
Edad:	<input type="text" value="57"/>

TAREAS

Tarea: Comprar leche

☐ Eliminar

Nombre:

Tarea: Sacar al perro

☐ Eliminar

Nombre:

Tarea: #3

Nombre:

Tarea: #4

Nombre:

+ Agregar Tarea adicional.

Lo que hemos hecho en el panel de administrador está muy bien, pero ¿sé podrá hacer en nuestras propias vistas? Veamos cómo se hace.

En nuestra aplicación ya tenemos un formulario para editar personas, es el siguiente:

```
view.py
def edit(request, persona_id):
    instancia = Persona.objects.get(id=persona_id)
    form = PersonaForm(instance=instancia)
    if request.method == "POST":
        form = PersonaForm(request.POST, instance=instancia)
        if form.is_valid():
            instancia = form.save(commit=False)
            instancia.save()
    return render(request, "core/edit.html", {'form': form})
```

Lo tengo perfectamente documentado en el otro tutorial y se ve de esta forma:

## Editar persona

Nombre:

Edad:

---

[Crear persona](#) | [Listar personas](#)

Nuestro objetivo es mostrar debajo los **inline** igual que hacemos en el panel de administrador.

Para ello necesitamos contar con un formulario para manejar las instancias de **Tarea** y luego registrarlo como componente para un **inline form**, algo que haremos haciendo uso de un modelo de Django llamado **inlineformset\_factory**:

```
forms.py
from django.forms import ModelForm
from django.forms.models import inlineformset_factory
from .models import Persona, Tarea

class PersonaForm(ModelForm):
    class Meta:
        model = Persona
        fields = ['nombre', 'edad']

class TareaForm(ModelForm):
    class Meta:
        model = Tarea
        fields = ['nombre']

# Aquí registramos nuestro inline formset
TareasInlineFormSet = inlineformset_factory(
    Persona, Tarea, form=TareaForm,
    extra=2, can_delete=True)
```

Básicamente le estamos diciendo que nuestro formset está formado por los modelos **Persona** haciendo de padre y **Tarea** de hijo, mostrándose estos últimos con el formulario **TareaForm** con dos huecos y la opción de borrar las tareas habilitada.

Ahora viene la parte importante, tenemos que hacer uso de este inline en la vista **edit**, recuperar sus datos y guardarlos cuando se reciben:

```
view.py
from .forms import PersonaForm, TareasInlineFormSet

def edit(request, persona_id):
    instancia = Persona.objects.get(id=persona_id)
    form = PersonaForm(instance=instancia)

    # Creamos el formset de tareas con los datos de la instancia
```

```

formset = TareasInlineFormSet(instance=instancia)

if request.method == "POST":
    form = PersonaForm(request.POST, instance=instancia)

    # Actualizamos también los datos del formset de tareas
    formset = TareasInlineFormSet(request.POST,
instance=instancia)

    if form.is_valid():

        instancia = form.save(commit=False)
        instancia.save()

        # Guardamos también el formset si es válido
        if formset.is_valid():
            formset.instance = instancia
            formset.save()

        # Actualizamos la pantalla del formulario
        return redirect(f'/edit/{instancia.id}')

# Si llegamos al final renderizamos el formulario y el formset
return render(
    request, "core/edit.html", {'form': form, 'formset': formset})

```

Como véis es cuestión de repetir lo mismo pero con el **formset** como si fuera otro formulario cualquiera.

En este punto nos faltaría sólo añadir renderizar los inlines en el HTML, lo podemos hacer recorriendo con un for el **formset** que enviamos, pues en realidad es una colección de subformularios:

```

edit.html
<form method="POST">
    {{ form.as_p }}
    {% csrf_token %}

    <h3>Lista de tareas</h3>

    {% for form_tarea in formset %}
        {{ form_tarea }} <br />
    {% endfor %}

    <!-- Este miniformulario maneja el inline -->
    {{ formset.management_form }}

    <br><button type="submit">Editar</button>
</form>

```

Es extremadamente importante renderizar el **management\_form**, ya que de forma oculta se encarga de manejar todos los subformularios del inline y sin él no funcionaría nada.

Sea como sea con esto deberíamos tener los inlines funcionando perfectamente en nuestro formulario de edición:

## Editar persona

Nombre:

Edad:

### Lista de tareas

Nombre:  Eliminar: ☐

Nombre:  Eliminar: ☐

---

[Crear persona](#) | [Listar personas](#)

## Django en Ubuntu Server con Nginx, Gunicorn y Supervisor

Hoy comparto con vosotros este tutorial para los interesados en aprender a desplegar Django en GNU/Linux de una forma cómoda y sencilla.

A continuación os resumo para qué sirve cada uno de los componentes que se utilizan en un despliegue genérico.

- **Gunicorn:** Green Unicorn es un servidor WSGI HTTP para Python (pre-fork de unicorn de ruby). Consume poco y es bastante rápido.
- **Nginx:** Es un servidor web/proxy inverso ligero de alto rendimiento y un proxy para protocolos de correo electrónico. Nos ayudará a servir ficheros estáticos.
- **Supervisor:** Es un gestor de procesos para Linux. Nos permitirá crear un proceso en segundo plano de nuestro servidor gunicorn.

### Requisitos

- Servidor con Nginx.
- Python 3.
- Proyecto Django ya listo para desplegar.
- Borrar del fichero **urls.py** que Django sirva los ficheros estáticos al desactivar el **DEBUG**.
- Tener configurada la variable con el directorio de los ficheros estáticos **STATIC\_ROOT** en el **settings.py**:

```
proyecto/settings.py
# Añadir esta línea abajo del todo dependiendo de vuestro directorio
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
```

Suponiendo que vuestro proyecto lo tendréis en **dominio.com** os recomiendo clonar el repositorio en la ruta **/var/www/dominio.com/**.

Ahora aseguraos de tener un fichero **requirements.txt** con las dependencias del proyecto en **/var/www/dominio.com/requirements.txt** y si no tenéis dependencias como mínimo que contenga a Django:

```
requirements.txt
django
```

Ahora vamos a instalar **Pipenv** para crear nuestro entorno virtual, para ello necesitamos **Pip** en **Python 3**:

```
sudo apt install python3-pip
pip3 install pipenv
```

El siguiente paso es crear el entorno e instalar las dependencias, fácil:

```
cd /var/www/dominio.com
pipenv install -r requirements.txt
```

Con esto ya deberíamos tener nuestro entorno creado, vamos a dejar anotada la ruta del **python** del entorno porque más adelante la necesitaremos, podemos consultar haciendo:

```
pipenv run which python
```

Os debería aparecer algo de este estilo dependiendo de vuestro usuario, que debería ser el administrador (aunque yo estoy haciendo el tutorial directamente con root):

```
/root/.local/share/virtualenvs/dominio.com-EZwa4jqa/bin/python
```

Dejadlo copiado en alguna parte.

En este punto deberíamos tener el proyecto funcionando, quizá tendréis que configurar una base de datos pero en eso no me voy a meter, en este tutorial daremos por hecho que usamos SQLite para hacerlo más sencillo:

```
pipenv run python manage.py migrate
```

Recopilamos los ficheros estáticos de las diferentes apps en el directorio static del proyecto (hay que hacerlo siempre que modifiquemos alguno), recordad tener configurada la variable **STATIC\_ROOT** tal como indico arriba en los requisitos. Esto es necesario para que **Nginx** pueda servirlos correctamente:

```
pipenv run python manage.py collectstatic
```

Cualquier comando que debáis ejecutar recordad hacerlo con **pipenv run** para hacer referencia al Python del entorno virtual.

Tenemos el proyecto de Django preparado pero necesitamos un servidor para manejarlo, para ello vamos a utilizar **gunicorn**.

```
cd /var/www/dominio.com
pipenv install gunicorn
```

Vamos a probar si se lanza correctamente desde la raíz, justo donde está el **manage.py** (el puerto podéis cambiarlo):

```
cd /var/www/dominio.com
pipenv run gunicorn proyecto.wsgi:application --bind=127.0.0.1:8000
```

Si se muestra el mensaje típico de **Listening at: http://127.0.0.1:8000** podemos hacer **Control + C** y confirmar que está funcionando bien.

El siguiente paso es mantener activo ese servidor de **gunicorn** en segundo plano, para ello usaremos el gestor de procesos **Supervisor**.

```
sudo apt install supervisor
```

Ahora tenemos que crear un fichero de configuración para nuestro proyecto:

```
sudo nano /etc/supervisor/conf.d/dominio.com.conf
```

En él añadiremos esta simple configuración, recuperando ya la ruta al **python** de nuestro entorno virtual, la que copiamos anteriormente (atención al identificador único de vuestro entorno virtual):

```
/etc/supervisor/conf.d/dominio.com.conf
[program:dominio.com]
command = /root/.local/share/virtualenvs/dominio.com-
EZwa4jqa/bin/python /root/.local/share/virtualenvs/dominio.com-
EZwa4jqa/bin/gunicorn proyecto.wsgi:application --bind=127.0.0.1:8000
directory = /var/www/dominio.com/proyecto
user = root
```

Es muy importante poner correctamente las rutas a **python** y a **gunicorn** del entorno virtual para crear correctamente el comando con los ejecutables de ambos programas en el entorno. El usuario tiene que ser el vuestro o **root**, pero debe tener los permisos adecuados. Además si queremos tener varios Django funcionando tendremos que ponerlos en puertos diferentes, por ejemplo 8000, 8001, 8002 o los que queramos.

Para manejar el proceso debemos actualizar los cambios y activar el proyecto en **supervisor**:

```
sudo supervisorctl reread
sudo supervisorctl update
sudo supervisorctl start dominio.com
```

Con esto deberíamos tener **gunicorn** ejecutando **Django** en el puerto 8000 del sistema, ya sólo nos queda configurar un server block de **Nginx** haciendo de proxy reverso para enlazarlo a un dominio/subdominio y servir los ficheros estáticos.

Todas las acciones para manejar el proceso que ejecuta nuestro proyecto se pueden encontrar en la [documentación de Supervisor](#). Los más comunes son:

- **reread**: Recarga las configuraciones de los procesos.
- **update**: Rearga las configuraciones y reinicia los procesos afectados.
- **start**: Para iniciar un proceso.
- **stop**: Para detener un proceso.
- **restart**: Para reiniciar un proceso, algo necesario para actualizar los cambios al modificar el proyecto de Django:

```
sudo supervisorctl restart dominio.com
```

En este punto deberías poder hacer una petición con **cURL** y ver que efectivamente os devuelve índice de vuestra página:

```
curl http://127.0.0.1:8000
```

Os voy a dejar la configuración genérica del sitio funcionando en un dominio en el puerto 80 no seguro y sirviendo los ficheros estáticos. Es vuestra tarea adaptarla y añadir un certificado tal como explico en el [curso de configuración básica de Ubuntu Server](#).

Primero vamos a crear un directorio para almacenar los **logs de nginx**:

```
cd /var/www/dominio.com
mkdir logs
```

Ahora creamos el **server block** del sitio:

```
sudo nano /etc/nginx/sites-available/dominio.com
```

Esta es la configuración básica:

```
/etc/nginx/sites-available/dominio.com
server {

    # Puerto y nombre
    listen 80;
    server_name dominio.com www.dominio.com;

    # Logs de nginx
    access_log /var/www/dominio.com/logs/nginx.access.log;
    error_log /var/www/dominio.com/logs/nginx.error.log;

    # Ficheros estáticos
    location /static/ {
        alias /var/www/dominio.com/static/;
        expires 365d;
    }

    # Proxy reverso del puerto 8000
    location / {
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
```

```
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_read_timeout 1m;
        proxy_connect_timeout 1m;
        proxy_pass http://127.0.0.1:8000;
    }
}
```

Guardamos y creamos un enlace simbólico:

```
cd /etc/nginx/sites-enabled
sudo ln -s ../sites-available/dominio.com
```

Finalmente reiniciamos nginx:

```
sudo service nginx restart
```

Si tenéis configurados los ficheros media en Django en el directorio **media** del proyecto, podéis añadir lo siguiente en la configuración (cortesía de RulezCore):

```
# Media
location /media/ {
    alias /var/www/dominio.com/media/;
}
```

Y ya deberíamos tener funcionando Django en <http://dominio.com> cargando correctamente los ficheros estáticos.

Para profundizar más sobre las opciones de configuración, como crear registros de errores y todo éso os dejo las documentaciones oficiales:

- Documentación de Gunicorn.
- Documentación de Nginx.
- Documentación de Supervisor.

¡Espero que os sirva!