

Proyecto Web Empresarial

Empezamos el segundo proyecto del curso a lo largo del cual crearemos una web empresarial. Se trata de una página de presentación para un negocio, donde se muestra información de la empresa, su catálogo de servicios y otras funcionalidades.

Vamos a echar un vistazo al frontend que os he preparado, el cual podéis descargar en los recursos de esta lección y descomprimir en el directorio CursoDjango.

Demo de la Web Empresarial y recursos

Puedes ver el proyecto final en el siguiente enlace: <http://webempresa.pythonanywhere.com/>. Recuerda que sólo los alumnos matriculados pueden descargar los diseños frontend preparados para realizar el curso paso a paso.

Como podéis observar tenemos un frontend para una cafetería: **La Caffettiera, L'AUTENTICO CAFFÈ D'ITALIA**. Si navegais por las distintas páginas notaréis que se trata de un frontend bastante bien acabado, da la sensación de que no hay mucho que hacer, pero creedme que este proyecto nos servirá para aprender un montón.

En todo desarrollo web lo primero es diferenciar las páginas estáticas de las dinámicas, es decir, por un lado las que su contenido no cambiará y por otro las que el cliente podrá manejar desde el panel de administrador.

En el caso de la Caffettiera, las páginas estáticas serían la "portada", "historia" y "visítanos". Son páginas que apenas cambiarán con el tiempo y nos permiten trabajar más la parte frontend para hacerlas más atractivas. Todas ellas las manejaremos en la clásica app **Core**, como en la web personal.

Respecto a las secciones dinámicas nos quedan Servicios y Blog.

Fijaros como la estructura de los Servicios es siempre la misma: subtítulo, título, imagen de fondo y un texto de contenido. Y por otro lado, las entradas del Blog también tienen una estructura común: fecha, título, imagen, texto de contenido, autor y categorías. Seguro que ya os habéis percatado de que ambas se pueden trasladar a modelos. Las manejaremos en sus respectivas apps **Services** y **Blog**.

Pero eso no es todo, hay otros elementos que nos conviene tener automatizados y que encontramos en el pie de página, me refiero a los enlaces a las redes sociales y las páginas genéricas con políticas de empresa, avisos legales, etc. Las redes sociales las vamos a gestionar en una app **Social** donde daremos al cliente la posibilidad de establecer el enlace a varias de ellas:

twitter, facebook, instagram, etc. Respecto a las páginas secundarias, tendremos una app **Pages** para gestionarla.

Finalmente tenemos la sección de contacto. Que podríamos considerarla una página estática, pero como tiene que capturar el formulario y enviarnos los mensajes en forma de email, la vamos a gestionar en su propia app.

En resumen, el backend de "La Caffettiera" estará formado por ni más ni menos que seis apps completamente reutilizables:

- **Core:** para gestionar las páginas estáticas (portada, historia y visítanos).
- **Services:** para gestionar los servicios de la sección servicios.
- **Blog:** para gestionar las entradas y sus categorías.
- **Social:** para manejar los enlaces a las redes sociales del pie de página.
- **Pages:** para gestionar las páginas secundarias del pie de página.
- **Contact:** para manejar capturar el formulario de contacto y enviar un email con el mensaje.

Preparar la app [Core]

Con el objetivo de practicar lo que hemos aprendido hasta ahora, tendrás que preparar el terreno para este segundo proyecto.

- Crea un proyecto Django de nombre webempresa utilizando nuestro entorno virtual django2.

```
(django2) django-admin startproject webempresa
```

- Añade una app Core con una vista para cada página de la cafetería, deberás añadir las respectivas URL y lograr que todo funcione. Por ahora puedes devolver un HttpResponse simple con el nombre de la página:

- **Inicio** home/
- **Historia** about/
- **Servicios** services/
- **Visítanos** store/
- **Contacto** contact/
- **Blog** blog/
- **Sample** sample/ (esta es para páginas de prueba)

```
(django2) python manage.py startapp core
```

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

```

def home(request):
    return HttpResponse("Inicio")

def about(request):
    return HttpResponse("Historia")

def services(request):
    return HttpResponse("Servicios")

def store(request):
    return HttpResponse("Visítanos")

def contact(request):
    return HttpResponse("Contacto")

def blog(request):
    return HttpResponse("Blog")

def sample(request):
    return HttpResponse("Sample")

core/urls.py
from django.contrib import admin
from django.urls import path
from core import views

urlpatterns = [
    path('', views.home, name="home"),
    path('about/', views.about, name="about"),
    path('services/', views.services, name="services"),
    path('store/', views.store, name="store"),
    path('contact/', views.contact, name="contact"),
    path('blog/', views.blog, name="blog"),
    path('sample/', views.sample, name="sample"),
    path('admin/', admin.site.urls),
]

```

Pongo el servidor en marcha y pruebo algunas urls a ver si funcionan:



Inicio



Sample

Listo.

Organizando mejor nuestras URLs

Antes de continuar con la siguiente práctica vamos a hacer un inciso. Quiero explicaros una forma de manejar mejor las URLs de nuestras apps.

Hasta ahora hemos visto como añadir todas nuestras URL únicamente en el fichero `urls` del proyecto, pero imaginarnos si tuviéramos varias apps cada una con sus urls, sería un lío tremendo, todo lleno de imports a las vistas de cada app.

Para solucionarlo podemos crear configuraciones URL específicas para cada app y luego importarlas en el fichero **`urls.py`** de nuestro proyecto bajo una url global. Esta es precisamente la forma como funcionan las URLs de la app **`admin`**, vamos a hacerlo con nuestra app **`core`**, ya veréis que fácil.

Vamos a empezar creando un nuevo fichero **`urls.py`** en ella y añadiremos dentro una nueva configuración de paths en una lista *`urlpatterns`*, igual que que la del **`urls.py`** del proyecto:

```
core/urls.py
from django.urls import path
from . import views

urlpatterns = [

]
```

Dentro vamos a trasladar los *`paths`* tal cual los tenemos, borrándolos de un lado para ponerlos en el otro:

```
core/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name="home"),
    path('about/', views.about, name="about"),
    path('services/', views.services, name="services"),
    path('store/', views.store, name="store"),
    path('contact/', views.contact, name="contact"),
    path('blog/', views.blog, name="blog"),
    path('sample/', views.sample, name="sample"),
]
```

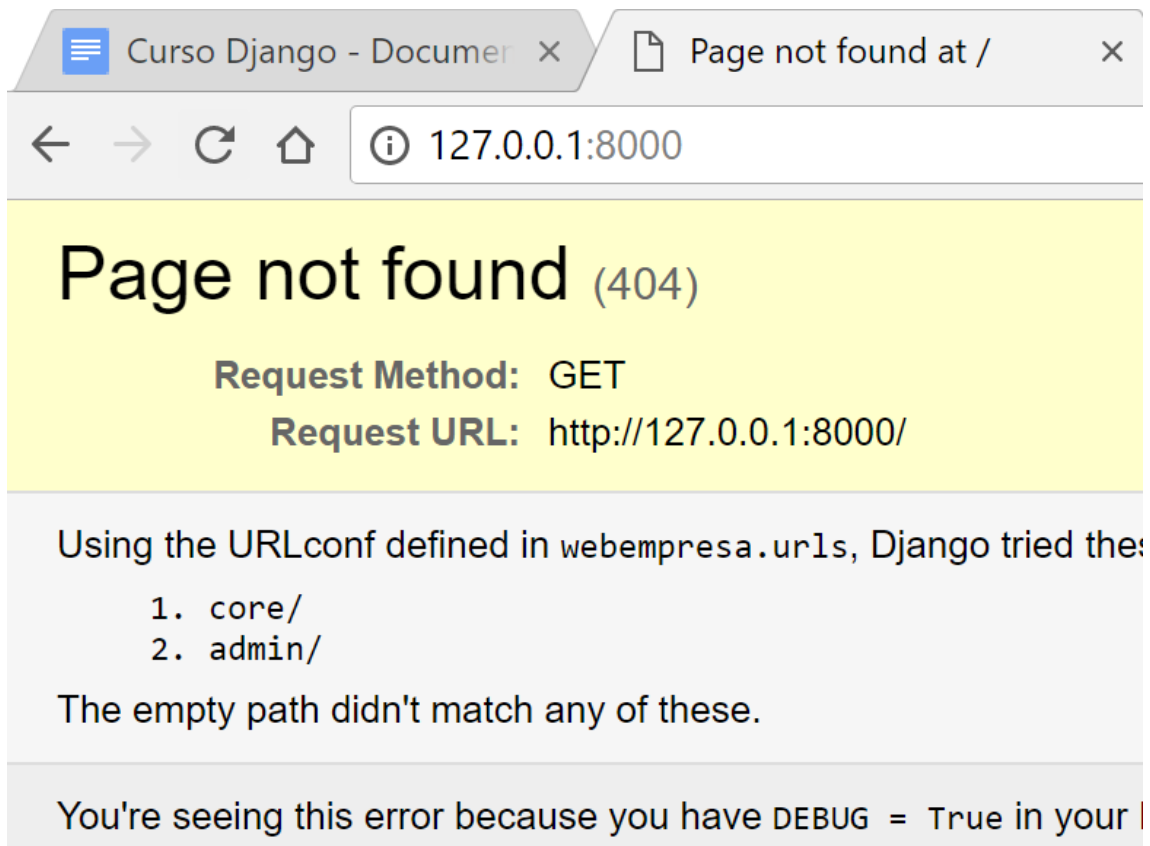
Ahora lo que haremos es importar estas URLs todas a la vez en el **`urls.py`** del proyecto bajo un path global. En el comentario de **`urls.py`** nos explican como se hace:

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

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

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

Vamos a probar en nuestro proyecto:



Como véis no funciona. ¿Por qué? Bueno fijaros que estamos definiendo un path global llamado core/ para incluir en él las URLs de la app **core**:

```
path('core/', include('core.urls')),
```

En otras palabras, todos nuestros Paths dependen de la raíz core/, representando el propio core/ la portada que tenemos en core:

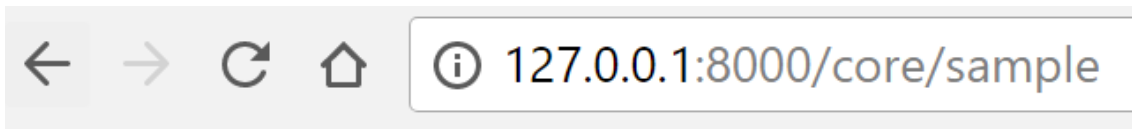


Inicio

Como este path no tiene una URL, representa la propia raíz:

```
path('', views.home, name='home'),
```

¿Como entraríamos en las otras páginas? Pues simplemente añadiendo después de core/ su path:



Sample

Por ahora no queremos esto, así que vamos a hacer que la raíz de las URLs de **core** esté vacía:

```
path('', include('core.urls')),
```

Así replicaremos el funcionamiento que teníamos antes:



Inicio

Y con esto ya sabéis manejar URLs de una forma mucho más cómoda.

Fusionar el Frontend y el Backend

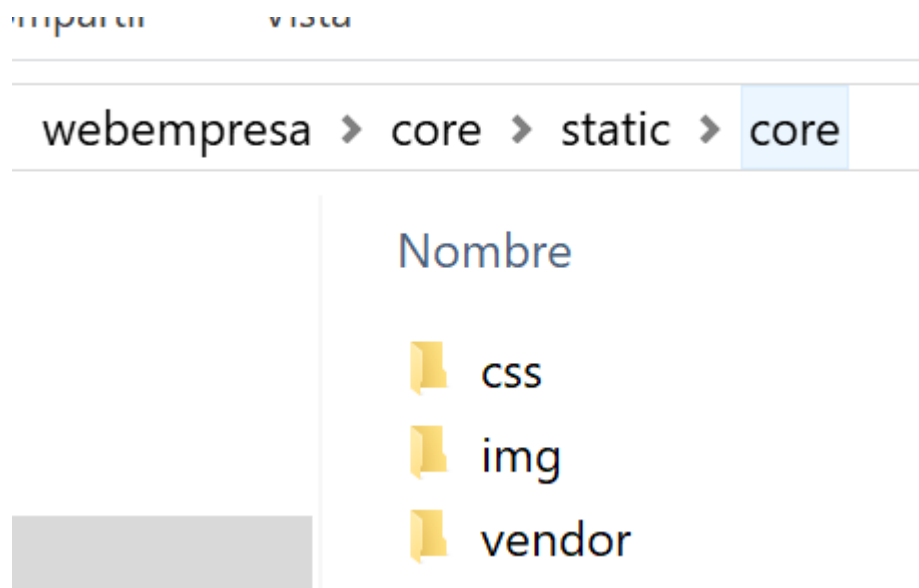
En esta lección deberás realizar la fusión. Identifica las partes comunes en todas las maquetas HTML y crea una estructura con herencia como hicimos con el primer proyecto (base.html, home.html...). Deberás lograr un menú funcional y que se carguen correctamente los ficheros estáticos (css, javascripts, imágenes) de todos los templates.

Nota

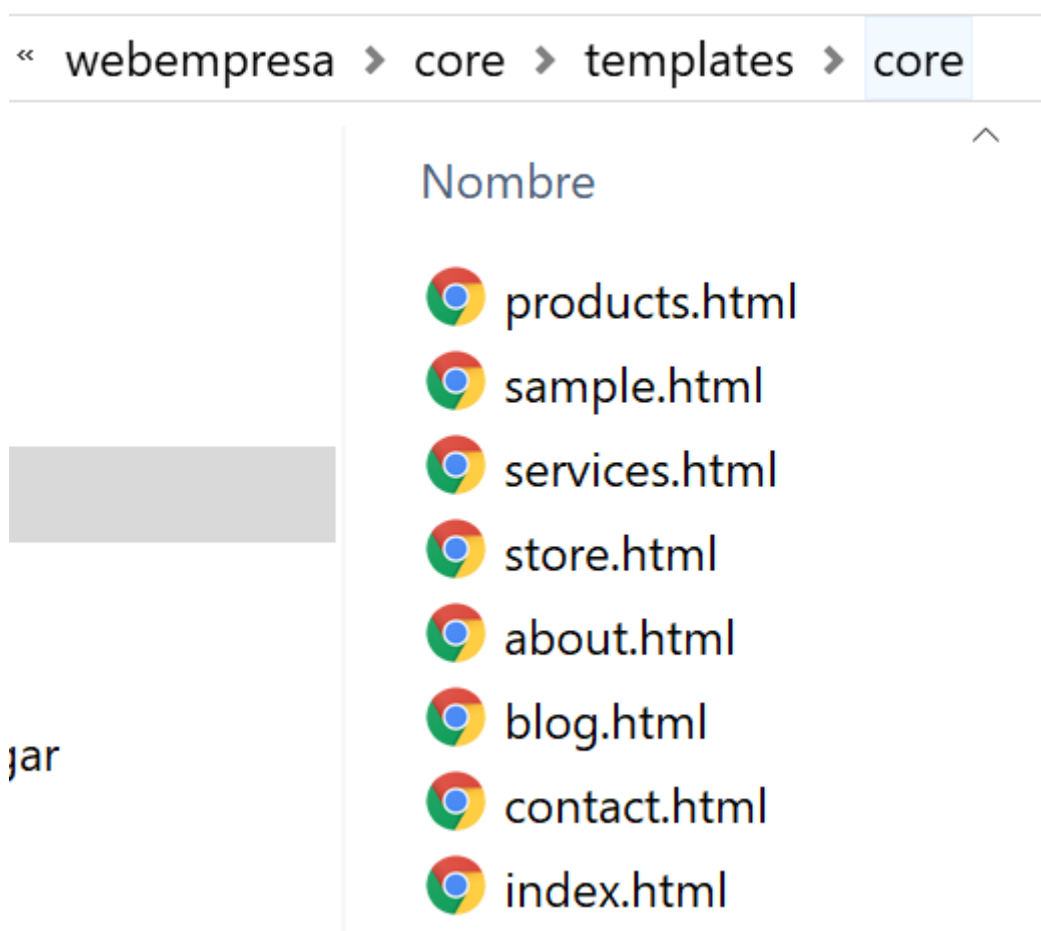
No olvides activar la app core y reiniciar el servidor para poder utilizar los recursos estáticos cargándolos con {% load static %} en su respectivo template.

Solución

Empezamos por ejemplo creando los directorio **static/core/** en nuestra app y copiamos ahí todos los recursos del frontend:



Ahora vamos a hacer lo mismo pero para los templates en el directorio **templates/core/**:

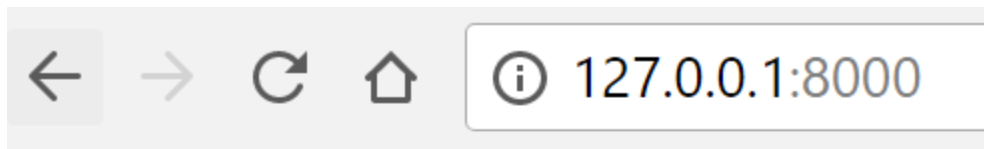


Ahora vamos a decirle a Django que cargue los ficheros estáticos y los templates de la app **core**, así que lo añadiremos en **settings.py**.

Acto seguido en lugar de seguir devolviendo un `HttpResponse` en las vistas vamos a renderizar nuestros templates. Voy a empezar por `home` y el template `index.html` al que cambiaré el nombre a `home.html` para que todo concuerde:

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

Ahora pruebo a ver si funciona:



L'autentico caffè c

[La Caffettiera](#)

- [Inicio](#)
- [Historia](#)
- [Servicios](#)
- [Visítanos](#)
- [Contacto](#)
- [Blog](#)

Parece que carga bien, pero los recursos estáticos no funcionan. Vamos a utilizar el template tag `load_static` y a adaptar sus urls en `home.html`:

```
core/templates/core/home.html
{% load static %}

<!-- Bootstrap -->
<link href="{% static 'core/vendor/bootstrap/css/bootstrap.min.css'
%}" rel="stylesheet">

<!-- Fuentes -->
<link href="{% static 'core/vendor/font-awesome/css/font-
awesome.min.css' %}" rel="stylesheet" type="text/css">
```



```

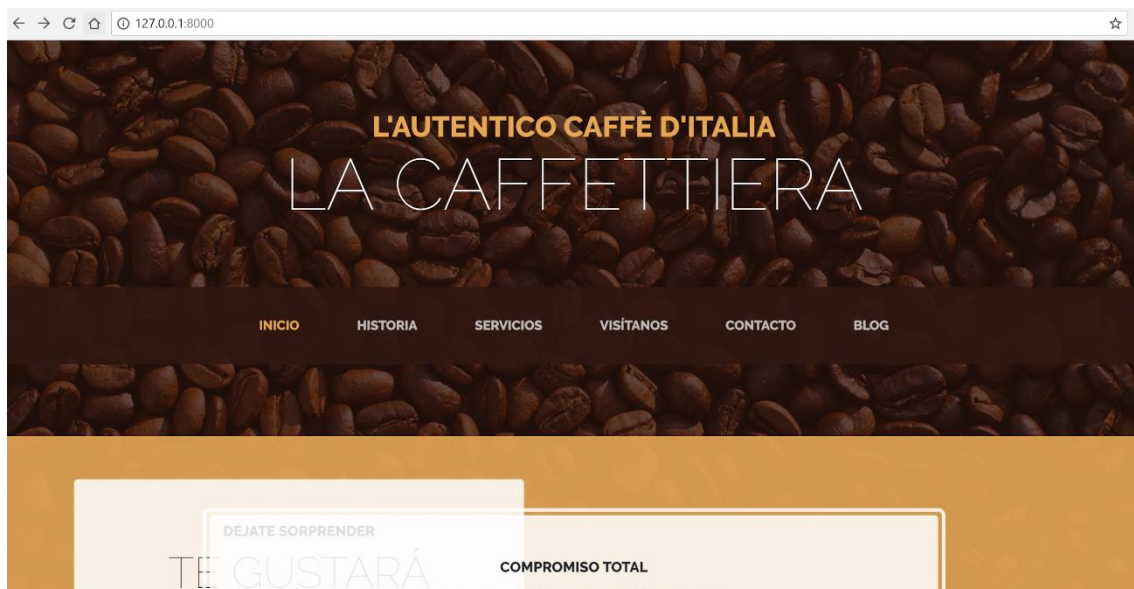
<link
href="https://fonts.googleapis.com/css?family=Raleway:100,100i,200,200
i,300,300i,400,400i,500,500i,600,600i,700,700i,800,800i,900,900i"
rel="stylesheet">
<link
href="https://fonts.googleapis.com/css?family=Lora:400,400i,700,700i"
rel="stylesheet">

<!-- Estilos -->
<link href="{% static 'core/css/business-casual.css' %}"
rel="stylesheet">

<!-- Bootstrap -->
<script src="{% static 'core/vendor/jquery/jquery.min.js'
%}"></script>
<script src="{% static
'core/vendor/bootstrap/js/bootstrap.bundle.min.js' %}"></script>

```

Probamos de nuevo:



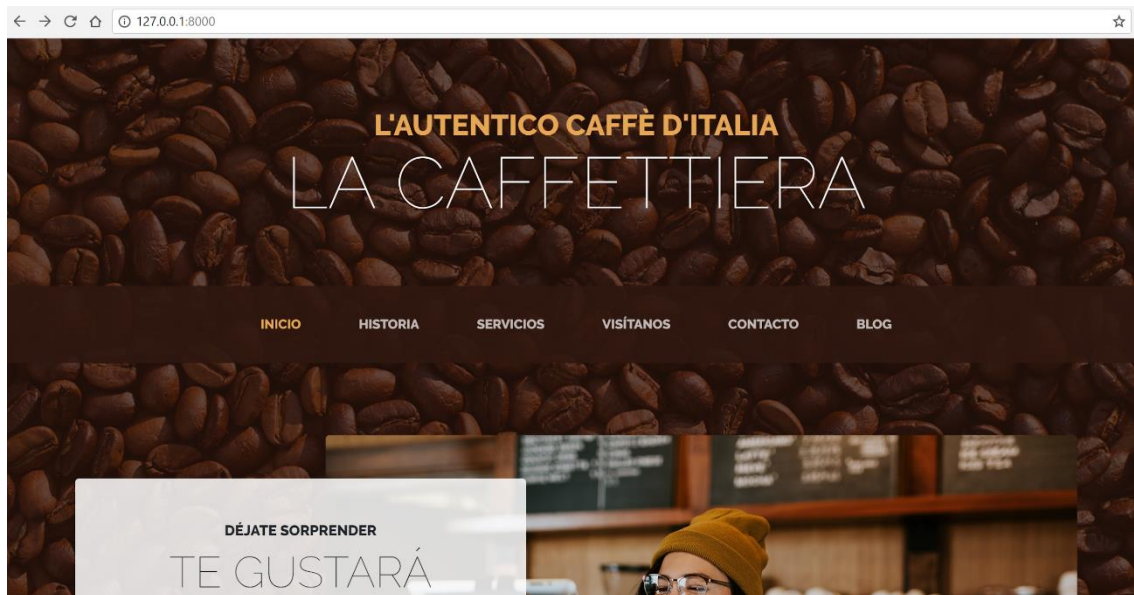
Más o menos funciona, pero como la imagen de la cabecera no se está mostrando todo se descuadra. Vamos a poner bien su enlace estático:

```

core/templates/core/home.html


```

Ahora sí:



Ahora que tenemos nuestra portada vamos a crear la plantilla base.html. Podemos simplemente copiar home.html y llamarla base.html porque lo tenemos todo ahí.

Vamos a echar un vistazo al diseño general de todas ellas para identificar la parte común y la parte que cambia...

Si nos fijamos un poco veremos que la estructura del contenido varía bastante entre ellas, así que lo más acertado en este caso sería crear un bloque content que abarque todo el espacio entre el menú y el pie:

```
core/templates/core/base.html
</nav>

{% block content %} {% endblock %}

<!-- Pié de página -->
<footer class="footer text-faded text-center py-5">
```

Ahora en cada template dejaremos únicamente esa parte dentro de un bloque content.

```
core/templates/core/base.html
{% extends 'core/base.html' %}

{% load static %}

{% block title %}Inicio{% endblock %}

{% block content %}
  <!-- Cabecera -->
  <section class="page-section clearfix">...</section>

  <!-- Mensaje -->
  <section class="page-section cta">...</section>
{% endblock %}
```

No debemos olvidar cargar el tag static en el template si para poder utilizarlo y en cuanto a los enlaces del menú vamos a utilizar el template tag url:

```
core/templates/core/base.html
<!-- Navegación -->
<a class="navbar-brand text-uppercase text-expanded d-lg-none"
    href="{% url 'home' %}">La Caffettiera</a>
...
<div class="collapse navbar-collapse" id="navbarResponsive">
    <ul class="navbar-nav mx-auto">
        <li class="nav-item px-lg-4">
            <a class="nav-link text-uppercase text-expanded"
                href="{% url 'home' %}">Inicio</a>
        </li>
        <li class="nav-item px-lg-4">
            <a class="nav-link text-uppercase text-expanded"
                href="{% url 'about' %}">Historia</a>
        </li>
        <li class="nav-item px-lg-4">
            <a class="nav-link text-uppercase text-expanded"
                href="{% url 'services' %}">Servicios</a>
        </li>
        <li class="nav-item px-lg-4">
            <a class="nav-link text-uppercase text-expanded"
                href="{% url 'store' %}">Visítanos</a>
        </li>
        <li class="nav-item px-lg-4">
            <a class="nav-link text-uppercase text-expanded"
                href="{% url 'contact' %}">Contacto</a>
        </li>
        <li class="nav-item px-lg-4">
            <a class="nav-link text-uppercase text-expanded"
                href="{% url 'blog' %}">Blog</a>
        </li>
    </ul>
</div>
```

Evidentemente para que funcionen bien tendremos que renderizar los templates bien fusionados. Voy a preparar primero todas las view:

```
core/views.py
from django.shortcuts import render

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

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

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

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

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

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

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

En cuanto a los templates los tengo preparados, voy a sustituirlos. Podéis descargarlos en los recursos de esta lección y hacer lo mismo. He incorporado elementos como el bloque title para mostrar un título dinámico en cada página.

Una vez vez tengamos todo listo deberíamos poder navegar correctamente entre las páginas y tendremos lista nuestra práctica.

Resaltando la sección actual

Uno de esos detalles que quizá pasa desapercibido en nuestro frontend es que dependiendo de la sección que visitamos, ésta nos aparece resaltada en el menú.

Según el código HTML de la maqueta para resaltar un elemento del menú hay que añadirle la clase active en su etiqueta `//`.

¿Se os ocurre alguna forma de manejar la página activa y resaltarla con esta clase? No sé si recordaréis la variable `request.path`, la utilizamos en el primer proyecto para mostrar o no una etiqueta `hr` que dibujaba una línea debajo del contenido en todas las páginas menos la portada. Bueno, pues podemos hacer lo mismo para establecer esta clase active:

```
core/templates/core/base.html  
<ul class="navbar-nav mx-auto">  
  <li class="nav-item px-lg-4"  
    {% if request.path == '/' %}active{% endif %}>  
    <a class="nav-link text-uppercase text-expanded"  
      href="{% url 'home' %}">Inicio</a>  
  </li>  
  <li class="nav-item px-lg-4"  
    {% if request.path == '/about/' %}active{% endif %}>  
    <a class="nav-link text-uppercase text-expanded"  
      href="{% url 'about' %}">Historia</a>  
  </li>  
  <li class="nav-item px-lg-4"  
    {% if request.path == '/services/' %}active{% endif %}>  
    <a class="nav-link text-uppercase text-expanded"  
      href="{% url 'services' %}">Servicios</a>  
  </li>  
  <li class="nav-item px-lg-4"  
    {% if request.path == '/store/' %}active{% endif %}>  
    <a class="nav-link text-uppercase text-expanded"  
      href="{% url 'store' %}">Visítanos</a>  
  </li>  
  <li class="nav-item px-lg-4"  
    {% if request.path == '/contact/' %}active{% endif %}>  
    <a class="nav-link text-uppercase text-expanded"  
      href="{% url 'contact' %}">Contacto</a>  
  </li>  
  <li class="nav-item px-lg-4"  
    {% if request.path == '/blog/' %}active{% endif %}>
```

```
<a class="nav-link text-uppercase text-expanded"
    href="{% url 'blog' %}">Blog</a>
</li>
</ul>
```

Y con esto lo tenemos:



No es que sea muy elegante porque si cambiamos la URL dejará de funcionar, pero es lo más fácil que podemos hacer para lograrlo.

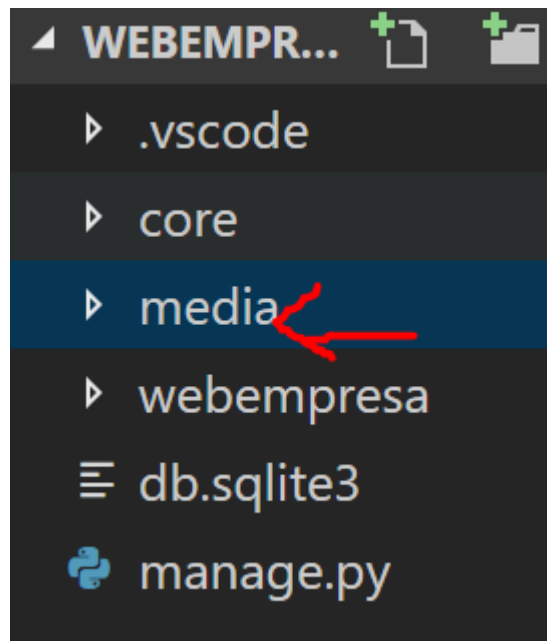
Tercera App [Services] Modelo y admin

La app Services es en esencia lo mismo que la app portafolio de la web personal así que estoy seguro de que podréis implementarla sin ayuda. Para hacerlo más llevadero la crearás en dos partes. Empezarás por crear el modelo y configurando el panel de administrador. Aquí tienes las indicaciones:

- Configura los ficheros media que funcionen en el servidor de desarrollo.
- Crea una app Services y añádela a la lista *INSTALLED_APPS* en *settings.py*.
- El modelo Service constará de 6 campos obligatorios, podéis utilizar de referencia el modelo Project de la app Portfolio del primer proyecto:
 - **Title:** Un título con 200 caracteres de longitud máxima.
 - **Subtitle:** Un subtítulo con 200 caracteres de longitud máxima.
 - **Content:** Un texto de tamaño indefinido.
 - **Image:** Una imagen para mostrar de fondo almacenada en el directorio media/services.
 - **Created:** Un campo automático para gestionar la fecha y hora de creación.
 - **Updated:** Un campo automático para gestionar la fecha y hora de última actualización.
- Haz una migración completa (makemigrations y migrate a secas) y crea un superusuario para poder acceder al panel de administrador.
- Ahora configura la app para ser manejable desde el panel de administrador. Debes mostrar los campos especiales Created y Updated en modo sólo lectura.
- Toda la información de la app deberá aparecer en español (admin en general, nombre de la app, nombre del modelo y sus campos).

Solución

Creamos el directorio media en la raíz del proyecto:



Añadimos la configuración *MEDIA* al **settings.py**:

```
webempresa/settings.py
# Media config
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

Configuramos el PATH en **webempresa/urls.py** para servir ficheros media en modo *DEBUG*:

```
webempresa/urls.py
from django.conf import settings

if settings.DEBUG:
    from django.conf.urls.static import static
    urlpatterns += static(settings.MEDIA_URL,
                          document_root=settings.MEDIA_ROOT)
```

Una vez preparados los ficheros media creamos la app y la añadimos a **settings.py**:

```
(django2) python manage.py startapp services
```

Creamos el modelo Service, podemos basarnos en el de Proyecto que hicimos para la app Portafolio:

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

class Service(models.Model):
    title = models.CharField(max_length=200,
                             verbose_name="Título")
    subtitle = models.CharField(max_length=200,
                                verbose_name="Subtítulo")
    content = models.TextField(
        verbose_name="Contenido")
    image = models.ImageField(verbose_name="Imagen",
```

```

        upload_to="services")
    created = models.DateTimeField(auto_now_add=True,
        verbose_name="Fecha de creación")
    updated = models.DateTimeField(auto_now=True,
        verbose_name="Fecha de edición")

    class Meta:
        verbose_name = "servicio"
        verbose_name_plural = "servicios"
        ordering = ['-created']

    def __str__(self):
        return self.title

```

Creamos las migraciones:

```

(django2) python manage.py makemigrations
(django2) python manage.py migrate

```

Creamos un superusuario:

```

(django2) python manage.py createsuperuser

```

Activamos la app en el **admin.py**:

```

services/admin.py
from django.contrib import admin
from .models import Service

class ServiceAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')

admin.site.register(Service, ServiceAdmin)

```

Ya podemos acceder al admin:



Finalmente tocaría acabar de traducir el nombre de la app y configurar django en español:

```

webempresa/settings.py
LANGUAGE_CODE = 'es'
services/apps.py
from django.apps import AppConfig

class ServicesConfig(AppConfig):
    name = 'services'

```

```
verbose_name = 'Gestor de servicios'
```

Activamos la configuración extendida:

```
webempresa/settings.py
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'core',
    'services.apps.ServicesConfig',
]
```

Y ya lo tenemos:



GESTOR DE SERVICIOS

Servicios

Tercera App [Services] Vista y template

En esta segunda parte te enfocarás en crear la vista y su respectivo template:

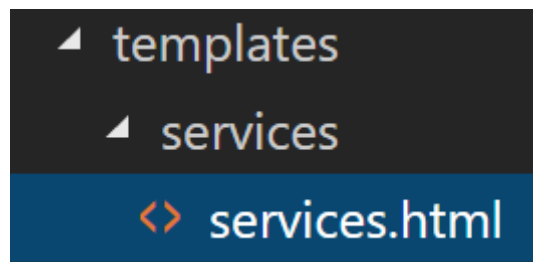
- Crea algunos servicios de prueba utilizando el panel de administrador, puedes utilizar los servicios del frontend y sus imágenes (directorio img) del frontend.
- Traslada el template services.html a un directorio templates/services en su propia app.
- Traslada la vista services a su propia app (no olvides borrar su path en las urls de core) y renderiza el template anterior. Llámala como quieras pero evita el nombre services ya que es redundante con el de la propia app y puede llevar a errores.
- Configura la vista que has creado en un fichero urls.py de app (services/urls.py) tal como hicimos con la app Core en las urls del proyecto, de manera que funcione en la url /services de la web.
- Finalmente fusiona el template para que muestre los servicios creados en el panel de administrador.

Solución

Básicamente voy a hacer un copiar y pegar de lo que tengo en el frontend:

<input type="checkbox"/>	SERVICIO
<input type="checkbox"/>	TAMBIÉN VENDEMOS
<input type="checkbox"/>	BOLLERÍA ARTESANAL
<input type="checkbox"/>	CAFÉS Y TÉS

Me llevo el template a su propio directorio de la app:



Me llevo la vista a la nueva app y muestro el nuevo

template: services/views.py

```
from django.shortcuts import render
```

```
def services(request):
```

```
    return render(request, "services/services.html")
```

Configuro las urls de app y en el proyecto (borro services de

core/urls): services/urls.py

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

```
from . import views
```

```
urlpatterns = [  
    path('', views.services, name="services"),  
]
```

webempresa/urls.py

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

Compruebo que todo funcione:



Finalmente realizo la fusión, tomando los servicios en la vista y pasándolos al template: `services/views.py`

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

def services(request):
    services = Service.objects.all()
    return render(request, "services/services.html",
                  {'services': services})
```

Y con un bucle for muestro todos los servicios siguiente la plantilla. Por defecto aparecen ordenados de más nuevos a más antiguos, pero (podemos usar el `reversed` en el propio template para voltear la lista, cuestión de gustos):

```
services/templates/services/services.html
{% extends 'core/base.html' %}

{% load static %}

{% block title %}Servicios{% endblock %}

{% block content %}

{% for service in services reversed %}
    <section class="page-section">
    <div class="container">
        <div class="product-item">
            <div class="product-item-title d-flex">
                <div class="bg-faded p-5 d-flex mr-auto rounded">
                    <h2 class="section-heading mb-0">
                        <span class="section-heading-upper">
                            {{service.subtitle}}
                        </span>
                        <span class="section-heading-lower">
                            {{service.title}}
                        </span>
                    </h2>
                </div>
            </div>
            
            <div class="product-item-description d-flex ml-auto">
                <div class="bg-faded p-5 rounded">
                    <p class="mb-0">{{service.content}}</p>
                </div>
            </div>
        </div>
    </div>
</div>
</div>
```

```
</section>
{% endfor %}

{% endblock %}
```

Con esto hemos completado nuestra tercera app del curso.

Cuarta App [Blog] Relaciones

Con la sección de Servicios lista nos toca desarrollar la app blog para que nuestros clientes puedan publicar noticias.

Mi idea es aprovechar el desarrollo de esta nueva app para introducir las relaciones entre modelos, una capacidad muy potente del sistema ORM de Django. Concretamente veremos dos tipos, las relaciones con ForeignKeys (claves foráneas) y las relaciones Many2Many (de muchos a muchos).

Las claves foráneas nos permiten enlazar una instancia de un modelo con otra instancia de otro modelo, o incluso del mismo. Esto es perfecto por ejemplo para enlazar una entrada con un usuario, representando éste al autor. En cambio con las relaciones de muchos a muchos, Many2Many para los amigos, podremos enlazar no sólo una instancia, sino varias, lo cual es muy conveniente para asignar varias categorías a una misma entrada de forma fácil y cómoda, ya veréis.

Vamos a empezar creando nuestra app **Blog**:

```
(django2) python manage.py startapp blog
```

Ahora creamos los dos modelos **Category** y **Post**, uno para las categorías y otra para las entradas. Como haremos uso del modelo Category en el modelo Post, lo declararemos primero.

```
blog/models.py
from django.db import models
from django.utils.timezone import now
from django.contrib.auth.models import User

class Category(models.Model):
    name = models.CharField(max_length=100,
        verbose_name="Nombre")
    created = models.DateTimeField(auto_now_add=True,
        verbose_name="Fecha de creación")
    updated = models.DateTimeField(auto_now=True,
        verbose_name="Fecha de edición")

    class Meta:
        verbose_name = "categoría"
        verbose_name_plural = "categorías"

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

```

class Post(models.Model):
    title = models.CharField(max_length=200,
                             verbose_name="Título")
    content = models.TextField(
        verbose_name="Contenido")
    published = models.DateTimeField(default=now,
                                     verbose_name="Fecha de publicación")
    image = models.ImageField(upload_to="blog", null=True, blank=True,
                              verbose_name="Imagen")
    author = models.ForeignKey(User, on_delete=models.CASCADE,
                               verbose_name="Autor")
    categories = models.ManyToManyField(Category,
                                       verbose_name="Categorías")
    created = models.DateTimeField(auto_now_add=True,
                                   verbose_name="Fecha de creación")
    updated = models.DateTimeField(auto_now=True,
                                   verbose_name="Fecha de edición")

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

    def __str__(self):
        return self.title

```

A comentar:

- El datetime **published** nos permite establecer una fecha manual de publicación de la entrada, así como la imagen que subiremos al directorio media/blog y es optativa.
- Por otro lado la relación ForeignKey haciendo referencia el primer parámetro al modelo User importado de **django.contrib.auth.models**, que maneja Django de forma automática y dentro suyo el parámetro **on_delete=models.CASCADE** que le indica a Django que si borramos un usuario, se borrarán también todas las entradas de las cuales sea el autor, de ahí lo de cascada, porque el modelo User se lleva los modelos relacionados con él.
- Finalmente la relación Many2Many apuntando al modelo Category y que nos permitirá seleccionar una o más categorías.

Ahora activamos la app en **settings.py**, creamos un admin básico y migramos:

```

blog/admin.py
from django.contrib import admin
from .models import Category, Post

class CategoryAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')

class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')

admin.site.register(Category, CategoryAdmin)
admin.site.register(Post, PostAdmin)

```

```
(django2) python manage.py makemigrations blog
(django2) python manage.py migrate blog
```

Con esto ya podemos entrar al admin y experimentar el potencial de las relaciones de nuestros modelos:

BLOG

Categorías

Entradas

Añadir entrada

Título:

Contenido:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec leo dui, vehicula vel dapibus ac, tempus non enim. Nunc tempor vel lacus vel gravida. Nam sit amet pellentesque mi. Aliquam eget porta mi, quis fermentum metus. Curabitur efficitur pellentesque tellus nec volutpat. In viverra mattis sem, facilisis condimentum mi rutrum ut. Quisque ut pellentesque dui. Nullam eu vehicula metus. Pellentesque id interdum elit. Aenean in efficitur enim.

Fecha de publicación: Fecha: Hoy Hora: Ahora

Nota: Usted esta a 1 horas por delante de la hora del servidor.

Imagen: products-01.jpg

Autor:

Categorías:

General

Ofertas

Mantenga presionado "Control" o "Command" en un Mac, para seleccionar más de una opción.

Podemos seleccionar un autor de entre los usuarios registrados en nuestra web gracias a la relación ForeignKey de 1 a muchos. Y varias categorías (incluso crearlas in-situ) aprovechando el potencial de las relaciones Many2Many de Muchos a Muchos:

Añadir categoría

Nombre:

General

Fecha de creación:

-

Fecha de edición:

-

No me diréis que no es genial, esta es una de las razones por las que me enamoré de Django.

Personalizando el administrador (2)

Toca un intermedio para seguir aprendiendo cómo personalizar el administrador. Hay un montón de cosas que podemos hacer con nuestros modelos, por ejemplo añadirles columnas de visualización, campos de ordenación y de búsqueda.

Vamos a practicar con las entradas de nuestro blog. Por defecto sólo nos aparece el título:

<input type="checkbox"/>	ENTRADA
<input type="checkbox"/>	OFERTAS DE OTOÑO

Pero podemos mostrar casi todos los campos. Vamos al fichero admin.py para personalizar qué columnas queremos mostrar.

```
blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published')
```

<input type="checkbox"/>	TÍTULO	AUTOR	FECHA DE PUBLICACIÓN
<input type="checkbox"/>	OFERTAS DE OTOÑO	hector	28 de Febrero de 2018 a las 22:43

Como véis así es mucho mejor, y además podemos ordenar por columnas.

También podemos indicar una tupla de ordenación, ésta indicará las prioridades de ordenación al mostrarse la tabla inicialmente.

```
blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published')
    ordering = ('author', 'published')
```

Esto nos agrupará las entradas por autor y las ordenará por fecha de publicación:

<input type="checkbox"/>	TÍTULO	AUTOR	1 ▲	FECHA DE PUBLICACIÓN	2 ▲
<input type="checkbox"/>	OFERTAS DE OTOÑO	hector		28 de Febrero de 2018 a las 22:43	

Si quisiéramos indicar sólo un campo de ordenación igualmente debéis crear una tupla con por lo menos un campo y una coma, sino Django no entenderá que es una tupla.

Otra cosa interesante es mostrar un formulario de búsqueda a partir de algunos campos. Es fácil y además podemos indicar campos de los modelos relacionados:

```
blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published')
    ordering = ('author', 'published')
    search_fields = ('title', 'content', 'author__username',
                    'categories__name')
```

Q

Acción:

<input type="checkbox"/>	TÍTULO	AUTOR
<input type="checkbox"/>	OFERTAS DE OTOÑO	hector

También cuando trabajamos con modelos que tengan campos de fechas y horas es posible activar el filtro avanzado con `date_hierarchy`, una imagen vale más que mil palabras:

```
blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published')
    ordering = ('author', 'published')
    search_fields = ('title', 'content', 'author__username',
                    'categories__name')
    date_hierarchy = 'published'
```

◀ 2018 28 de Febrero

Y ya que hablamos de filtros no podemos olvidar la tupla `list_filter`, gracias a la cual podemos agrupar por campos directamente en la tabla:

```
blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published')
    ordering = ('author', 'published')
    search_fields = ('title', 'content', 'author__username',
                    'categories__name')
    date_hierarchy = 'published'
    list_filter = ('author__username', 'categories__name')
```


FILTRO

Por nombre de usuario

Todo

hector

Por Nombre

Todo

General

Ofertas

o ya para poner la guinda al pastel, quizá os estáis preguntando si es posible mostrar la lista de categorías:

```
blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published', 'categories') #
    <==
    ordering = ('author', 'published')
    search_fields = ('title', 'content', 'author__username',
                    'categories__name')
    date_hierarchy = 'published'
    list_filter = ('author__username', 'categories__name')
```

Si lo probáis veréis que no se pueden añadir campos Many2Many en la opción **list_display**, pero no os preocupéis, os voy a enseñar a crear vuestros propios campos.

Lo que vamos a hacer es definir dentro de la clase PostAdmin un método cuyo nombre debe concuerde con el de la nueva columna que queremos crear. Además de self le pasaremos obj, que representa ni más ni menos que el objeto de cada fila que se muestra en la tabla del administrador. Esto lo gestiona Django así que no le deis muchas vueltas:

```
blog/admin.py
```

```

class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published', 'categories') #
<====
    ordering = ('author', 'published')
    search_fields = ('title', 'content', 'author__username',
                    'categories__name')
    date_hierarchy = 'published'
    list_filter = ('author__username', 'categories__name')

    def post_categories(self, obj):
        pass

```

Ahora tenemos que devolver una cadena de texto con el valor a mostrar. Nosotros queremos una lista de categorías, así que vamos a crear una cadena con los nombres de todas las categorías que tiene el objeto accediendo a su campo `categories.all()` mientras las ordenamos por nombre:

```

blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published', 'categories')
    ordering = ('author', 'published')
    search_fields = ('title', 'content', 'author__username',
                    'categories__name')
    date_hierarchy = 'published'
    list_filter = ('author__username', 'categories__name')

    def post_categories(self, obj):
        return ", ".join(
            [c.name for c in obj.categories.all().order_by("name")])

```

Finalmente añadiremos este método a nuestra tupla `list_display` como si fuera una columna más a mostrar:

```

blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published', 'post_categories')
# <==
    ordering = ('author', 'published')
    search_fields = ('title', 'content', 'author__username',
                    'categories__name')
    date_hierarchy = 'published'
    list_filter = ('author__username', 'categories__name')

    def post_categories(self, obj):
        return ", ".join(
            [c.name for c in obj.categories.all().order_by("name")])

```

Y ahí lo tenemos:

POST CATEGORIES

General, Ofertas

Ya seguro estáis pensando...¿Pero que hacemos con el nombre de la columna? ¿Cómo lo cambiamos? Pues sólo tenemos que modificar el atributo **short_description** del método de esta forma:

```
blog/admin.py
class PostAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'author', 'published', 'post_categories')
    ordering = ('author', 'published')
    search_fields = ('title', 'content', 'author__username',
                    'categories__name')
    date_hierarchy = 'published'
    list_filter = ('author__username', 'categories__name')

    def post_categories(self, obj):
        return ", ".join(
            [c.name for c in obj.categories.all().order_by("name")])
    post_categories.short_description = "Categorías"
```

CATEGORÍAS

General, Ofertas

Con esto acabamos el intermedio. Si os interesa saber cómo generar código HTML en lugar de simple texto os dejo un enlace a StackOverflow donde lo explican muy bien haciendo uso de la función `mark_safe`:

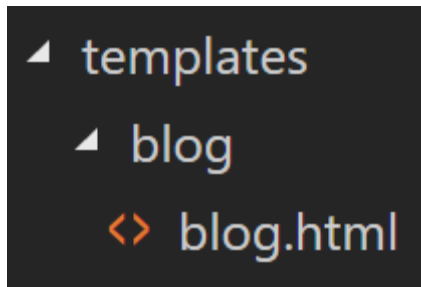
<https://stackoverflow.com/questions/47953705/how-do-i-use-allow-tags-in-django-2-0-admin>

En la próxima lección empezaremos a crear las vistas de nuestras app Blog.

Creando las vistas del Blog (1)

Bien, ya tenemos los modelos listos así que nos toca desarrollar las vistas. Vamos a crear vistas, una para mostrar todas las noticias, y otra para filtrar por categorías. En el próximo proyecto os enseñaré a crear paginadores y otras cosas interesantes, pero por ahora lo vamos a mantener lo más simple posible.

Vamos a prepararlo todo, trasladando el template `blog.html` y su respectiva vista a la app Blog y poniendo bien las urls:



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

def blog(request):
    return render(request, "blog/blog.html")
blog/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.blog, name="blog"),
]
webempresa/urls.py
urlpatterns = [
    path('', include('core.urls')),
    path('blog/', include('blog.urls')),
    path('services/', include('services.urls')),
    path('admin/', admin.site.urls),
]
```

Ya debería funcionar:



Vamos a recuperar las noticias y a fusionar su template:

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

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

Fusionaremos todo menos las categorías:

```
blog/templates/blog/blog.html
{% extends 'core/base.html' %}

{% load static %}
```

```

{% block title %}Blog{% endblock %}

{% block content %}
  {% for post in posts %}
    <section class="page-section cta">
      <div class="container">
        <div class="row">
          <div class="col-xl-9 mx-auto">
            <div class="cta-innerv text-center rounded">
              <h2 class="section-heading mb-5">
                <span class="section-heading-upper">
                  {{post.published}}</span>
                <span class="section-heading-lower">
                  {{post.title}}</span>
              </h2>
              <p class="mb-0">
                
              </p>
              <p class="mb-0 mbt">{{post.content}}</p>
              <p class="mb-0 mbt">
                <span class="section-heading-under">
                  Publicado por
                  <em><b>{{post.author}}</b></em>
                  en <em></em>
                </span>
              </p>
            </div>
          </div>
        </div>
      </section>
    {% endfor %}
  {% endblock %}

```

Si echamos un vistazo al resultado todo está correcto, fijaros como el autor hace referencia al Usuario y nos lo muestra bien:

Publicado por **hector** en *General, Ofertas*

Sin embargo la fecha de publicación no está en el formato que necesitamos, según nuestro frontend sólo necesitamos mostrar DIA/MES/AÑO separados con barras. Para conseguir este resultado podemos utilizar el template tag date y darle el formato deseado. En nuestro caso nos interesa uno predeterminado llamado **SHORT_DATE_FORMAT**:

```

<span class="section-heading-upper">
  {{post.published|date:"SHORT_DATE_FORMAT"}}
</span>

```

28/02/2018

Otra cosa importante es que por defecto no se respetan los saltos de línea en el cuerpo de la noticia, para activarlos debemos indicar el template tag **linebreaks** en el contenido:

```
<p class="mb-0 mbt">
    {{post.content|linebreaks}}
</p>
```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec leo dui, vehicula vel dapibus ac, tempus non enim. Nunc tempor vel lacus vel gravida. Nam sit amet pellentesque mi. Aliquam eget porta mi, quis fermentum metus. Curabitur efficitur pellentesque tellus nec volutpat. In viverra mattis sem, facilisis condimentum mi rutrum ut.

Quisque ut pellentesque dui. Nullam eu vehicula metus. Pellentesque id interdum elit. Aenean in efficitur enim.

Y por último vamos a por las categorías. Si comentamos lo que tenemos en el diseño y simplemente mostramos la relación many2many nos aparecerá algo muy raro:

```
<p class="mb-0 mbt">
    <span class="section-heading-under">
        Publicado por <em><b>{{post.author}}</b></em>
        en <em>{{post.categories}}</em>
    </span>
</p>
```

Publicado por **hector** en blog.Category.None

Esto es porque categories se comporta como una consulta a la base de datos, tenemos que indicarle exactamente qué queremos mostrar, normalmente añadiendo **.all** para en nuestro caso hacer referencia a todas las categorías de la entrada:

```
<p class="mb-0 mbt">
    <span class="section-heading-under">
        Publicado por <em><b>{{post.author}}</b></em>
        en <em>{{post.categories.all}}</em>
    </span>
</p>
```

Publicado por **hector** en <QuerySet [<Category: General>, <Category: Ofertas>]>

Esto hará la consulta y nos devolverá un QuerySet, evidentemente eso no lo queremos. Podemos hacer algo más fácil, utilizar un templatetag join para mostrar las categorías separadas por comas:

```
<p class="mb-0 mbt">
    <span class="section-heading-under">
        Publicado por <em><b>{{post.author}}</b></em>
        en <em>{{post.categories.all|join:", "}}</em>
    </span>
</p>
```

Publicado por **hector** en General, Ofertas

Por ahora vamos a dejarlo así, luego cuando tengamos la vista para filtrar por categoría volveremos para poner los enlaces.

Creando las vistas del Blog (2)

Para las categorías tenemos que aplicar una lógica diferente de la que hemos utilizado hasta ahora. Lo primero es idear una forma de mostrar cada categoría, digamos que cada una es independiente de las otras y tendrá sus propias entradas, así que tenemos que diferenciarlas en la vista.

En estos casos la lógica más simple es enviar al Path el id del objeto que queremos recuperar. En nuestro caso podemos hacerlo de esta forma:

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

def blog(request):
    posts = Post.objects.all()
    return render(request, "blog/blog.html", {'posts':posts})

def category(request, category_id):
    pass
```

Para configurar la URL simplemente añadiremos un parámetro **category_id**. Antes de Django 2.0 este proceso requería utilizar expresiones regulares, pero ahora gracias a la función **path** es mucho más sencillo, sólo debemos indicar el nombre del parámetro entre <> (más pequeño, más grande):

```
blog/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.blog, name="blog"),
    path('category/<category_id>/', views.category,
name="category"),
]
```

Esto ya funcionará, pero por defecto **category_id** será una cadena y el campo id es un número entero. Podemos forzar la conversión a entero cambiándolo a:

```
path('category/<int:category_id>/', views.category, name="category"),
```

Sea como sea con esto añadimos dinamismo en la URL, de manera que podemos enviar un parámetro y nos será muy fácil recuperar la categoría, pero antes vamos a crear un template **category.html** como una copia de **blog.html**, sólo que en el título mostraremos el nombre de la categoría:

```
blog/templates/blog/category.html
```

```

{% extends 'core/base.html' %}

{% load static %}

{% block title %}{{category}}{% endblock %}

{% block content %}
    {% for post in posts %}
        <section class="page-section cta">
            <div class="container">
                <div class="row">
                    <div class="col-xl-9 mx-auto">
                        <div class="cta-innerv text-center rounded">
                            <h2 class="section-heading mb-5">
                                <span class="section-heading-upper">
                                    {{post.published}}</span>
                                <span class="section-heading-lower">
                                    {{post.title}}</span>
                            </h2>
                            <p class="mb-0">
                                
                                </p>
                                <p class="mb-0 mbt">{{post.content}}</p>
                                <p class="mb-0 mbt">
                                    <span class="section-heading-under">
                                        Publicado por
                                <em><b>{{post.author}}</b></em>
                                    en <em> </em>
                                </span>
                                </p>
                            </div>
                        </div>
                    </div>
                </div>
            </div>
        </section>
    {% endfor %}
{% endblock %}

```

Ahora en la vista simplemente recuperamos la categoría utilizando el método get de objects, pasándole el campo que tiene usar como filtro:

```

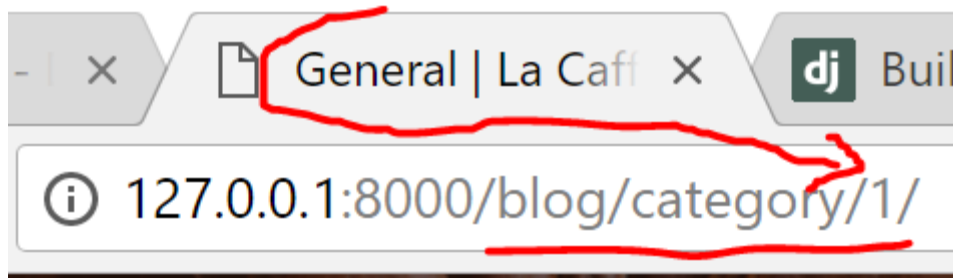
blog/views.py
from django.shortcuts import render
from .models import Post, Category

def blog(request):
    posts = Post.objects.all()
    return render(request, "blog/blog.html", {'posts':posts})

def category(request, category_id):
    category = Category.objects.get(id=category_id)
    return render(request, "blog/category.html",
{'category':category})

```

Si intentamos acceder a una categoría, por ejemplo la de id 1 que es la primera que se crea, nos la devolverá correctamente:



Lo malo de utilizar directamente el método `get`, es que si no se encuentra un resultado Django devolverá un error:



Para evitar esta situación y devolver un error 404, el típico de no encontrado, podemos utilizar un shortcut llamado **`get_object_or_404`**:

```
blog/views.py
from django.shortcuts import render, get_object_or_404
from .models import Post, Category

def blog(request):
    posts = Post.objects.all()
    return render(request, "blog/blog.html", {'posts':posts})

def category(request, category_id):
    category = get_object_or_404(Category, id=category_id)
    return render(request, "blog/category.html",
{'category':category})
```

Ahora si da error, por lo menos será un error dentro de la nomenclatura, ya que el 404 es el error que por norma hay que devolver si no se encuentra una página. Aunque con el Debug nos salga en forma de información:



Sea como sea ahora que tenemos la categoría podemos buscar sus entradas. Una forma rudimentaria de hacerlo es crear otra consulta para recuperar las entradas filtrando por categoría:

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

def blog(request):
    posts = Post.objects.all()
    return render(request, "blog/blog.html", {'posts':posts})

def category(request, category_id):
    posts = Post.objects.filter(categories=category)
    category = Category.objects.get(id=category_id)
    return render(request, "blog/category.html",
        {'category':category, 'posts':posts})
```

Esto ya nos funcionará:



Sin embargo como os decía es una forma rudimentaria de hacerlo, y eso es porque Django nos ofrece una forma mucho más fácil de hacerlo gracias a la capacidad de las relaciones de hacer consultas inversas.

Vamos a dejar la vista como la teníamos, únicamente pasando la categoría:

```
blog/views.py
from django.shortcuts import render, get_object_or_404
from .models import Post, Category

def blog(request):
    posts = Post.objects.all()
    return render(request, "blog/blog.html", {'posts':posts})

def category(request, category_id):
    category = get_object_or_404(Category, id=category_id)
    return render(request, "blog/category.html",
        {'category':category})
```

Ahora vamos a nuestro template, y en lugar de recorrer las entradas, que no tenemos porque no estamos pasando ninguna clave con este nombre, haremos la siguiente magia:

```
{% for post in category.posts_set.all %}
```

Si comprobamos nuestra web veremos que increíblemente funciona:

28/02/2018

OFERTAS DE OTOÑO

¿Cómo puede ser? Pues fácil, las relaciones no sólo existen en un sentido, sino en ambos. Aprovechando ésto, Django implementa una sintaxis genérica con **modelo.modeloRelacionado.set.all** para consultar todas las instancias del modeloRelacionado con el modelo.

Su limitación es que sólo podemos tener una relación a dos bandas con el mismo nombre genérico, pero no es nada que no se pueda arreglar manualmente. Por ejemplo en nuestro caso, simplemente deberíamos ir a nuestro modelo Post y en la relación Many2Many categories, añadir un campo llamado **related_name**, nombre relacionado:

```
blog/models.py
categories = models.ManyToManyField(Category,
    verbose_name="Categorías", related_name="get_posts")
```

Una vez hecho, en lugar del **_set.all** podemos llamar a este nombre relacionado:

```
{% for post in category.get_posts.all %}
```

Ahora que tenemos la página de categorías vamos a finalizar la app añadiendo los enlaces que dejamos pendientes. Por desgracia ya no podemos utilizar el join porque necesitamos crear un enlace, así que vamos a usar un for:

```
<span class="section-heading-under">
    Publicado por <em><b>{{post.author}}</b></em> en
    {% for category in post.categories.all %}
        <a href="{% url 'category' category.id %}" class="link">
            {{category.name}}</a>
    {% endfor %}
</span>
```

Antes de continuar no olvides añadir los enlaces también en el template **blog.html**.

Por último un pequeño detalle que quizá os pasará desapercibido, pero nuestro menú deja de resaltar BLOG cuando filtramos por una categoría:



Claro, la url ahora ya no es **/blog/** si no **/blog/category/etc**. Por suerte con un poco de ingenio podemos arreglarlo recortando los primeros caracteres del path con el filtro slice (que hace lo mismo que el slicing con las colecciones):

```
core/templates/core/base.html
<li class="nav-item px-lg-4
    {% if request.path|slice:"":6" == '/blog/' %}active{% endif %}">
    <a class="nav-link text-uppercase text-expanded"
        href="{% url 'blog' %}">Blog</a>
</li>
```

Y con esto damos por finalizada esta pequeña app Blog.

Quinta App [Social] Redes Sociales

En esta lección toca crear la app Social para permitirle a nuestros clientes configurar sus redes sociales.

```
(django2) python manage.py startapp social
social/models.py
from django.db import models

class Link(models.Model):
    key = models.SlugField(
        verbose_name="Nombre clave", max_length=100, unique=True)
    name = models.CharField(
        verbose_name="Red social", max_length=200)
    url = models.URLField(
        verbose_name="Enlace", max_length=200, null=True, blank=True)
    created = models.DateTimeField(
        verbose_name="Fecha de creación", auto_now_add=True)
    updated = models.DateTimeField(
        verbose_name="Fecha de edición", auto_now=True)

    class Meta:
        verbose_name = "enlace"
        verbose_name_plural = "enlaces"
        ordering = ['name']

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

Al modelo lo llamaremos link (enlace) y constará de tres sencillos campos: una clave de tipo Slug la cual nos obligará a utilizar caracteres alfanuméricos, guiones o barras; y que utilizaremos para consultar el registro a modo de diccionario, un nombre para la red social y una dirección URL. Es interesante comentar que sólo cambiando la url por un texto podríais conseguir una especie de panel de opciones configurables por el cliente, dándole la opción de cambiar cadenas de caracteres por ejemplo con el título de la página, metadatos o cualquier otro campo que podáis mostrar en el template.

A comentar también que como no puede haber dos enlaces para la misma red social, haremos que la clave sea única. Además aunque nosotros demos al cliente la posibilidades añadir varias redes sociales, es posible que algunas no las necesite, por lo que dejaremos el enlace como optativo.

Vamos a cambiarle el **verbose_nombre** a "Redes sociales", que siempre quedará mejor. La activamos y la migramos:

```
social/apps.py
from django.apps import AppConfig

class SocialConfig(AppConfig):
    name = 'social'
    verbose_name = 'Redes sociales'
```

Activamos la app con la configuración extendida:

```
webempresa/settings.py
'social.apps.SocialConfig',
```

Migramos:

```
(django2) python manage.py makemigrations social
(django2) python manage.py migrate social
```

Creamos un admin simple:

```
social/admin.py
from django.contrib import admin
from .models import Link

# Register your models here.
class LinkAdmin(admin.ModelAdmin):
    pass

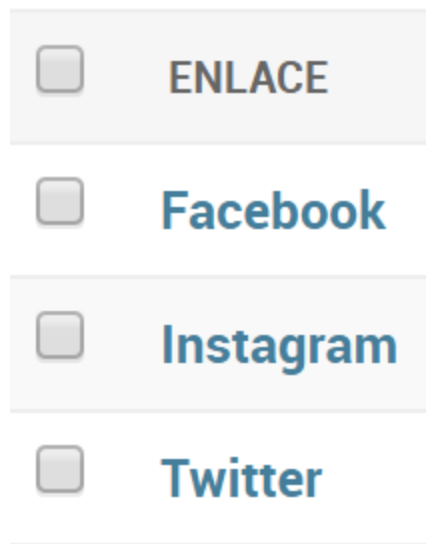
admin.site.register(Link, LinkAdmin)
```

REDES SOCIALES

Enlaces

Y algunas redes básicas:

LINK_TWITTER	Twitter	twitter.com
LINK_FACEBOOK	Facebook	facebook.com
LINK_INSTAGRAM	Instagram	instagram.com



A vertical list of four options, each consisting of a small square checkbox followed by a text label. The options are: 'ENLACE' (in all caps), 'Facebook' (with a capital 'F'), 'Instagram' (with a capital 'I'), and 'Twitter' (with a capital 'T'). The labels are in a blue font, while the checkboxes are light gray.

Vosotros podéis añadir las que queráis, aunque tened en cuenta que si son muy raras tal vez la librería Fontawesome no tenga un icono para mostrarlas.

Sea como sea ya tenemos los enlaces en la base de datos, ahora sólo tenemos que mostrarlas en nuestra web. Lo más sencillo sería ir a cada vista, recuperar las redes y enviarlas a sus respectivos templates... Pero está claro que eso no es óptimo ni elegante, de hecho es redundante, difícil de mantener y de extender. En su lugar aprenderemos algo mejor, pero lo veremos en la siguiente lección.

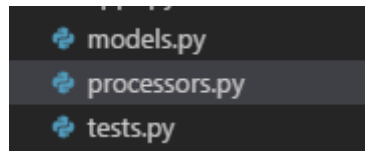
Procesadores de contexto

Así que tenemos que recuperar los enlaces sociales para enviarlos a todas las páginas... ¿Habrá alguna forma que nos permita hacerlo una vez y que funcione en todos los templates? ¡Pues sí! De hecho hay más de una, pero para este caso la más óptima es crear un procesador de contexto.

¿Qué es un procesador de contexto? Pues es una forma de extender el contexto, aunque de poco sirve decirlo si no sabemos antes qué es el contexto.

¿Recordáis el diccionario que enviamos desde nuestras vistas a los templates? Bueno, ese diccionario lo que hace es extender el contexto, por lo que podemos entender que se trata de una especie de diccionario común, que existe incluso sin enviar ningún dato desde una vista. Así que blanco y en botella, si logramos extender ese contexto global y añadir los enlaces de nuestras redes, entonces podremos mostrarlas en cualquier template sin necesidad de enviarlas desde una vista.

Para crear un procesador de contexto vamos a crear un nuevo fichero llamado **processors.py** en nuestra app Social:



Dentro vamos a definir una función que devuelva un diccionario de la siguiente forma:

```
social/processors.py
def ctx_dict(request):
    ctx = {'test': 'hola'}
    return ctx
```

Nuestro objetivo es que este diccionario extienda el contexto global, de manera que podamos utilizar la clave 'test' como una variable en cualquier template. Para lograrlo debemos ir a **settings**, buscar el apartado **context_processors** en el diccionario *TEMPLATES* dentro de la clave *OPTIONS* y añadirlo al final:

```
webempresa/settings.py
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
                'social.processors.ctx_dict' # <====
            ],
        },
    ],
]
```

Si ahora vamos por ejemplo a base.html y mostramos la variable `{{test}}` en el pie de página:

```
core/templates/core/base.html
<footer class="footer text-faded text-center py-5">
    <div class="container">
        {{test}}
        ...
    </div>
</footer>
```

Si actualizamos la web, veremos que aparece "Hola" en todas las páginas.

Por tanto ya tenemos la base, sólo debemos añadir al diccionario las redes:

```
social/processors.py
from .models import Link

def ctx_dict(request):
    ctx = {}
    links = Link.objects.all()
```

```

for link in links:
    ctx[link.key] = link.url
return ctx

```

Y mostrarlas en el template:

```

core/templates/core/base.html
<p class="m-0">
    {% if LINK_TWITTER %}
    <a href="{%LINK_TWITTER%}" class="link">
        <span class="fa-stack fa-lg">
            <i class="fa fa-circle fa-stack-2x"></i>
            <i class="fa fa-twitter fa-stack-1x fa-inverse"></i>
        </span>
    </a>
    {% endif %}
    {% if LINK_FACEBOOK %}
    <a href="{%LINK_FACEBOOK%}" class="link">
        <span class="fa-stack fa-lg">
            <i class="fa fa-circle fa-stack-2x"></i>
            <i class="fa fa-facebook fa-stack-1x fa-inverse"></i>
        </span>
    </a>
    {% endif %}
    {% if LINK_INSTAGRAM %}
    <a href="{%LINK_INSTAGRAM%}" class="link">
        <span class="fa-stack fa-lg">
            <i class="fa fa-circle fa-stack-2x"></i>
            <i class="fa fa-instagram fa-stack-1x fa-inverse"></i>
        </span>
    </a>
    {% endif %}
</p>

```

Sin duda una técnica extensible y elegante, digna de unos buenos profesionales.

Sexta App [Pages] Gestor de Páginas

En este punto tenemos nuestro proyecto bastante encaminado, sólo nos falta desarrollar la app para gestionar páginas de contenido secundario (políticas, avisos legales...) y la de contacto para manejar el formulario de contacto.

La app Pages es muy sencilla, pues la estructura de una página consta únicamente de un título y un contenido. Vamos a crearla:

```
(django2) python manage.py startapp pages
```

Traducimos el nombre:

```

pages/apps.py
from django.apps import AppConfig

class PagesConfig(AppConfig):
    name = 'pages'
    verbose_name = 'Gestor de páginas'

```


Añadimos la configuración extendida:

```
webempresa/settings.py
'pages.apps.PagesConfig',
```

Y ahora el modelo Page:

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

class Page(models.Model):
    title = models.CharField(max_length=200,
        verbose_name="Título")
    content = models.TextField(
        verbose_name="Contenido")
    created = models.DateTimeField(auto_now_add=True,
        verbose_name="Fecha de creación")
    updated = models.DateTimeField(auto_now=True,
        verbose_name="Fecha de edición")

    class Meta:
        verbose_name = "página"
        verbose_name_plural = "páginas"
        ordering = ['title']

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

Creamos un admin básico:

```
pages/admin.py
from django.contrib import admin
from .models import Page

class PageAdmin(admin.ModelAdmin):
    pass

admin.site.register(Page, PageAdmin)
```

Migramos:

```
(django2) python manage.py makemigrations pages
(django2) python manage.py migrate pages
```

GESTOR DE PÁGINAS

Páginas

☐ PÁGINA

☐ Aviso legal

☐ Cookies

☐ Política de privacidad

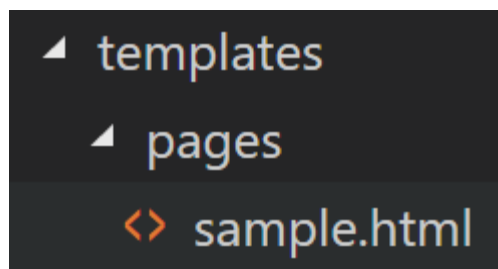
Perfecto, ahora tenemos que desarrollar las vistas. Seguiremos una lógica parecida a la del blog y sus categorías, de manera que estas páginas secundarias tengan el path **/page/<page_id>**.

Vamos a hacerlo:

```
pages/views.py
from django.shortcuts import render, get_object_or_404
from .models import Page

def page(request, page_id):
    page = get_object_or_404(Page, id=page_id)
    return render(request, 'pages/sample.html', {'page':page})
```

Ahora movemos el template sample.html en la propia app de páginas y **borramos la view y url sample de Core**:



Creamos las urls:

```
pages/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('<int:page_id>/', views.page, name="page"),
]

webempresa/urls.py
urlpatterns = [
    path('', include('core.urls')),
```

```

    path('blog/', include('blog.urls')),
    path('page/', include('pages.urls')),
    path('services/', include('services.urls')),
    path('admin/', admin.site.urls),
]

```

Probamos si nos aparece alguna página con id 1:



Perfecto, parece que funciona. Vamos a realizar la fusión, respetando los saltos de línea:

```

pages/templates/pages/sample.html
{% extends 'core/base.html' %}

{% load static %}

{% block title %}{{page.title}}{% endblock %}

{% block content %}
<section class="page-section about-heading">
  <div class="container">
    <div class="about-heading-content mbtm">
      <div class="row">
        <div class="col-xl-9 col-lg-10 mx-auto">
          <div class="bg-faded rounded p-5 forced">
            <h2 class="section-heading mb-4">
              <span class="section-heading-lower">
                {{page.title}}</span>
              </h2>
            <div class="section-content">
              {{page.content|linebreaks}}
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
</section>
{% endblock %}

```

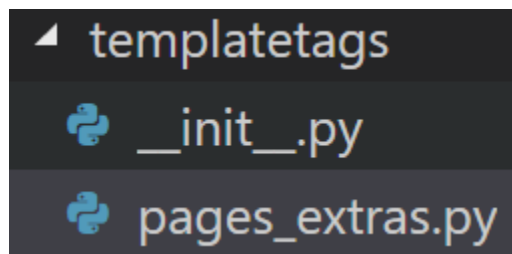


Lo tenemos listo, ya sólo falta mostrar los enlaces de las páginas en la parte inferior. Podríamos hacerlo añadiéndolas a un procesador de contexto como con las redes sociales, pero os comenté que se podía hacer de otra forma, os lo explico en la próxima lección.

Creando un Template Tag para listar páginas

En esta lección vamos a aprender a crear un template tag para mostrar contenido personalizado, concretamente lo que hará es recuperar la lista de páginas secundarias y devolverla. Es una alternativa más flexible que extender el procesador de contexto, pero también consume algo más de recursos.

Para crear nuestro propio template tag debemos seguir unos pasos. El primer es crear un directorio llamado **template tags** dentro de la app donde queremos añadir esta funcionalidad. En nuestro caso **pages/templatetags**. Dentro añadiremos un **init**, esto indicará a Python que se trata de un package, y justo al lado crearemos un script para almacenar nuestros template tags, podemos llamarlo como queramos pero yo siguiendo el ejemplo oficial de la documentación le voy a llamar **pages_extras.py**.



Ahora vamos a declarar el template tag, para ello debemos registrarlo en la librería de Templates, así que empezaremos importando el módulo de registro de templates:

```
pages/templatetags/pages_extras.py
from django import template
from pages.models import Page

register = template.Library()
```

El tag que vamos a crear es un relativamente simple. No necesitará que le pasemos ningún parámetro, simplemente devolverá la lista de páginas:

```
pages/templatetags/pages_extras.py
from django import template
from pages.models import Page

register = template.Library()

@register.simple_tag
def get_page_list():
    pages = Page.objects.all()
    return pages
```

Ahora muy importante, reiniciamos el servidor para que incluya los nuevos template tags en la memoria, y de vuelta a template **base.html** si cargamos

los **pages_extras** con `{% load %}`, ya deberíamos ser capaces de ejecutar el template tag:

```
core/templates/core/base.html
{% load pages_extras %}
{% get_page_list %}
```

`<QuerySet [<Page: Aviso legal>, <Page: Cookies>, <Page: Política de privacidad>]>`

Ahí tenemos nuestro QuerySet con las páginas, pero claro... así no podemos manejarlo, necesitamos tenerlo en una variable. No es difícil, sólo tenemos que darle un nombre al template tag de la siguiente forma:

```
core/templates/core/base.html
{% get_page_list as page_list %}
```

Una vez ejecutada esta línea podemos recorrer `page_list` con un `for` y mostrar la lista de páginas con sus respectivos enlaces:

```
core/templates/core/base.html
{% load pages_extras %}
{% get_page_list as page_list %}
{% for page in page_list %}
    <a href="{% url 'page' page.id %}" class="link">{{page.title}}</a>
    {% if not forloop.last %}·{% endif %}
{% endfor %}
```

Aviso legal · Cookies · Política de privacidad

Si queréis darle un toque más interesante a los enlaces de las páginas, siempre podéis pasar el título en forma de slug:

```
core/templates/core/base.html
{% load pages_extras %}
{% get_page_list as page_list %}
{% for page in page_list %}
    <a href="{% url 'page' page.id page.title|slugify %}"
class="link">
    {{page.title}}</a> {% if not forloop.last %}·{% endif %}
{% endfor %}
```

Aunque sólo es de adorno tendréis que definir un slug de mentira en el path y la vista:

```
`pages/urls.py`
``python
path('<int:page_id>/<slug:page_slug>/', views.page, name="page"),
pages/views.py
def page(request, page_id, page_slug):
```

Pero el resultado es interesante y puede mejorar el SEO de las páginas:

127.0.0.1:8000/page/2/aviso-legal/

En fin, con esto habéis aprendido otra forma de inyectar datos comunes en todas las páginas, sólo imaginad la de cosas que podéis llegar a a hacer.

Ordenación y edición de páginas

Si algo tiene el desarrollo de software es que nunca hay límite de mejora. Con eso en mente hay un par de funcionalidades que podemos añadir a nuestro sistema de páginas secundarias.

La primera consiste en un sistema de ordenación manual. Hasta ahora hemos estado ordenando nuestros modelos automáticamente, pero quizá algún cliente quiera establecer un orden distinto.

Cuando necesitemos hacerlo, os sugiero crear un campo en el modelo llamado order (orden) y de tipo Small Integer (entero pequeño) y con un valor 0 por defecto.

```
pages/models.py
order = models.SmallIntegerField(verbose_name="Orden", default=0)
```

También cambiaremos la ordenación por defecto al campo order, y de segunda opción el título:

```
pages/models.py
ordering = ['order', 'title']
```

Aplicamos las migraciones:

```
(django2) python manage.py makemigrations pages
(django2) python manage.py migrate pages
```

Y modificamos el admin.py para mostrar el orden en segundo lugar:

```
pages/admin.py
class PageAdmin(admin.ModelAdmin):
    readonly_fields = ('created', 'updated')
    list_display = ('title', 'order')
```

De vuelta a nuestro panel de administrador podemos establecer un número de orden a cada página, un orden que sin cambiar nada se respetará en el frontend:

<input type="checkbox"/>	TÍTULO	2 ▲	ORDEN
<input type="checkbox"/>	Política de privacidad		0
<input type="checkbox"/>	Cookies		1
<input type="checkbox"/>	Aviso legal		2

Política de privacidad · Cookies · Aviso legal

Sólo le tenemos que decir al cliente, que si quiere cambiar el orden debe poner un número y que los más pequeños se muestran antes.

El segundo detalle tiene que ver con la edición directa. Por ejemplo, tenemos al administrador de la página viendo la página de "políticas de privacidad" y detecta un fallo. Pues podríamos añadir un enlace de edición si la web detecta que un usuario ha iniciado sesión. Vamos a hacerlo.

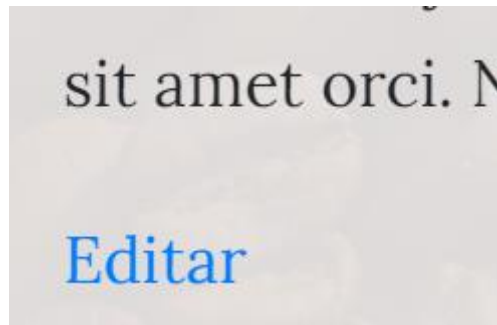
Para saber si un usuario está identificado, os gustará saber que Django inyecta información de la sesión activa en el contexto, igual que hicimos nosotros manualmente para las redes sociales. Concretamente lo maneja con:

```
webempresa/settings.py
'django.contrib.auth.context_processors.auth'
```

Como este procesador de contexto viene activado por defecto, podemos ir al template `sample.html` de las páginas y añadir el siguiente template tag:

```
pages/templates/pages/sample.html
<div class="section-content">
  {{page.content|safe}}
  {% if user.is_authenticated %}
    <p><a href="#">Editar</a></p>
  {% endif %}
</div>
```

Sí, en todos los templates tenemos una variable `user` que almacena el usuario identificado, y podemos comprobar cómodamente si se encuentra autenticado en el panel de administrador con su método **`is_authenticated`** para mostrar el enlace de edición.



Sólo tenemos que configurar el enlace, ¿a dónde debe apuntar? Obviamente a la dirección donde podemos editar la página. Podríamos ponerla en crudo, pero no es elegante. Lo haremos bien, utilizando `{% url %}` de la siguiente forma:

```
pages/templates/pages/sample.html
<div class="section-content">
    {{page.content|safe}}
    {% if user.is_authenticated %}
        <p><a href="{% url 'admin:pages_page_change' page.id
    %}">Editar</a></p>
    {% endif %}
</div>
```

127.0.0.1:8000/admin/pages/page/3/change/

La lógica es simple, `{admin}:{app}_{modelo}_{accion} {id_del_objeto}` (cuando sea necesario). Hay otras acciones a parte de change que la url de edición, como add y delete para añadir y borrar respectivamente.

Personalizando el administrador (3)

Antes de ponernos la última app de contacto y acabar la web, voy a enseñaros cómo añadir un editor WYSIWYG para los campos de texto de nuestro modelos.

Un editor WYSIWYG , del inglés "What You See Is What You Get" o "Lo que ves es lo que consigues", representa una interfaz avanzada para editar texto enriquecido, como si de un fichero Word se tratase.

Existen muchos editores WYSIWYG, pero uno de los más utilizados en el mundo del desarrollo web es CKEditor, un proyecto muy maduro y compatible con Django. Su integración es tan simple que en unos pocos pasos lo tendremos funcionando.

Empezaremos instalando la app django-ckeditor con pip:

```
(django2) pip install django-ckeditor
```

Luego la añadiremos a las apps de Django en **settings.py**:

```
webempresa/settings.py
INSTALLED_APPS = [
```

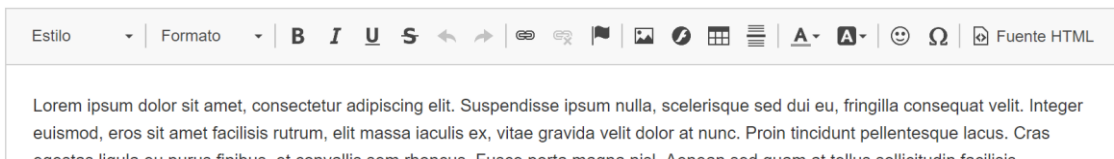


```
...
    'ckeditor',
]
```

Con esto ya tenemos listo CKEditor, sólo nos resta configurar los campos donde queramos mostrar el editor WYSIWYG. Lo haremos de la siguiente forma, importando el campo **RichTextField** de la app **CKeditor**, y sustituyendo el campo **models.TextField** por este que hemos importado:

```
pages/models.py
content = RichTextField(verbose_name="Contenido")
```

Ahora pondremos el servidor en marcha, vamos a editar una página y...



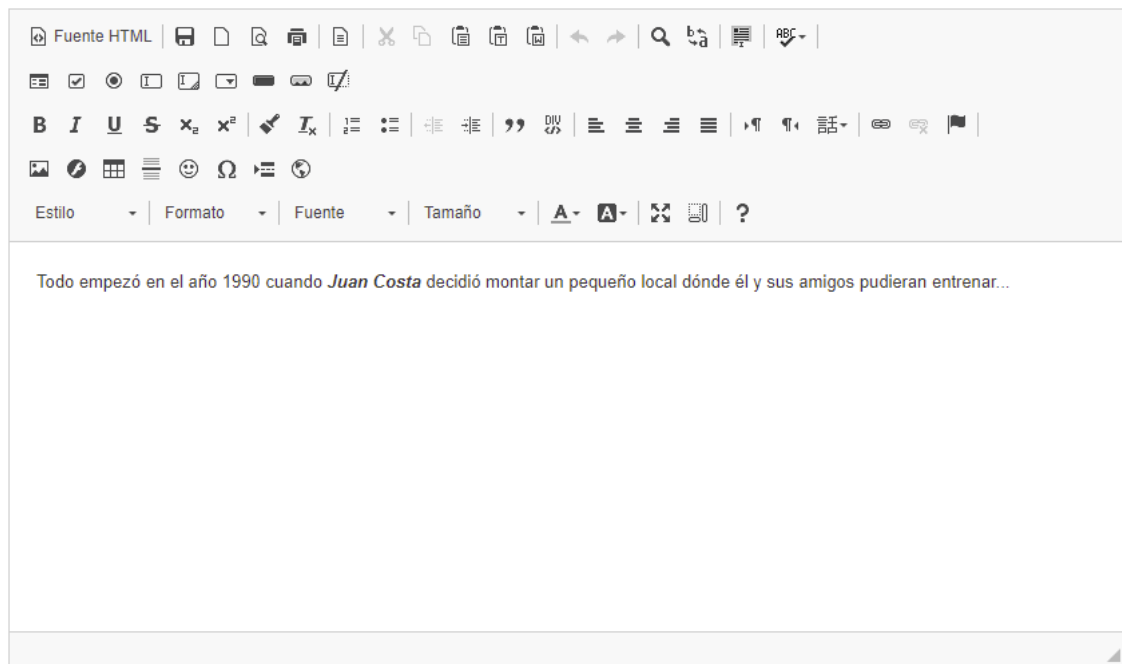
¿Qué os parece? Aquí tenemos un editor visual para nuestro campo de texto. En él podemos poner estilos visuales, negritas, cursivas, subrayados, enlaces, etc.

CKEditor incluye muchísimas funcionalidades y podríamos estar hablando de él horas y horas, así que por mi parte sólo os voy a enseñar como cambiar la configuración básica.

Para establecer una configuración personalizada en la barra de CKEditor tendremos que redefinir su diccionario de configuración en **settings.py**:

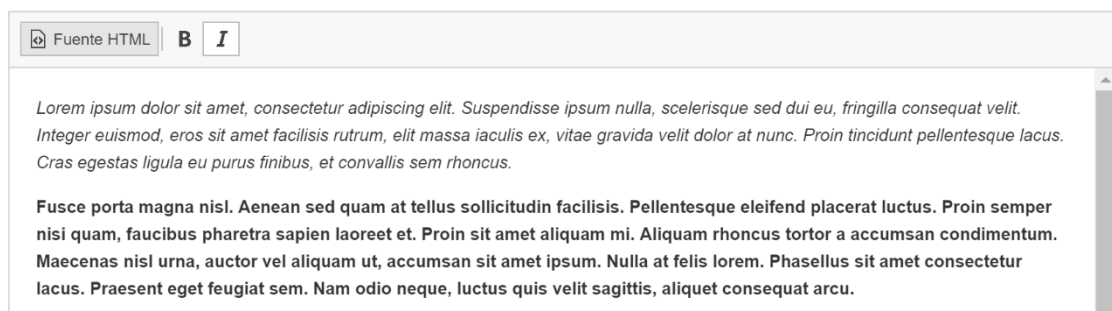
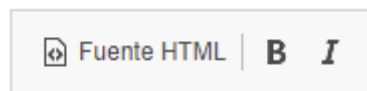
```
webempresa/settings.py
# Ckeditor
CKEDITOR_CONFIGS = {
    'default': {
        'toolbar': None,
    }
}
```

Si establecemos el valor de la toolbar en None le estaremos diciendo que muestre todos los campos posibles:



Esto es una bestialidad, por eso quizá os interesa más mostrar una versión compacta sólo con negritas y cursivas:

```
webempresa/settings.py
# Ckeditor
CKEDITOR_CONFIGS = {
    'default': {
        'toolbar': 'Basic',
    }
}
```



En la página de la app django-ckeditor encontraréis la documentación completa y ejemplos para adaptar la barra a vuestras necesidades: <https://github.com/django-ckeditor/django-ckeditor>

Por ejemplo una barra personalizada con nos dan de ejemplo la definiríamos de la siguiente forma:

```
webempresa/settings.py
# Ckeditor
CKEDITOR_CONFIGS = {
```

```

        'default': {
            'toolbar': 'Custom',
            'toolbar_Custom': [
                ['Bold', 'Italic', 'Underline'],
                ['NumberedList', 'BulletedList', '-', 'Outdent', 'Indent',
                '-',
                    'JustifyLeft', 'JustifyCenter', 'JustifyRight',
                'JustifyBlock'],
                ['Link', 'Unlink'],
                ['RemoveFormat', 'Source']
            ]
        }
    }
}

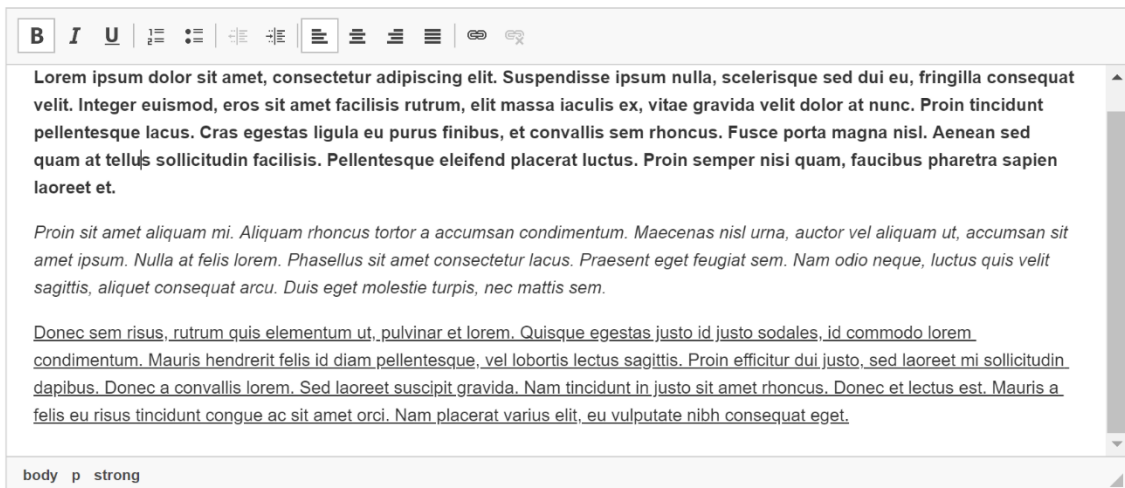
```

A partir de esta configuración podemos observar que cada sublista de toolbar_Custom es un grupo de botones, sería cuestión de jugar con ellos para mostrarlos y esconderlos:

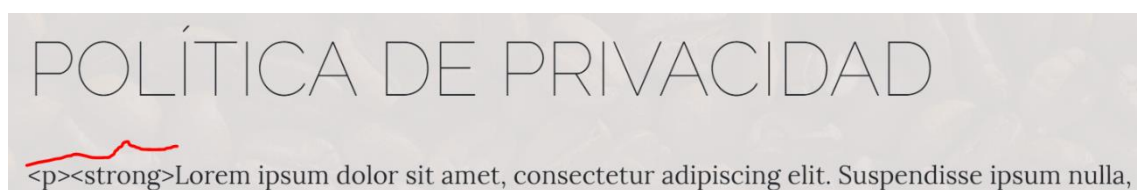
```

webempresa/settings.py
# Ckeditor
CKEDITOR_CONFIGS = {
    'default': {
        'toolbar': 'Custom',
        'toolbar_Custom': [
            ['Bold', 'Italic', 'Underline'],
            ['NumberedList', 'BulletedList', '-', 'Outdent', 'Indent',
            '-',
                'JustifyLeft', 'JustifyCenter', 'JustifyRight',
            'JustifyBlock'],
            ['Link', 'Unlink']
        ]
    }
}

```

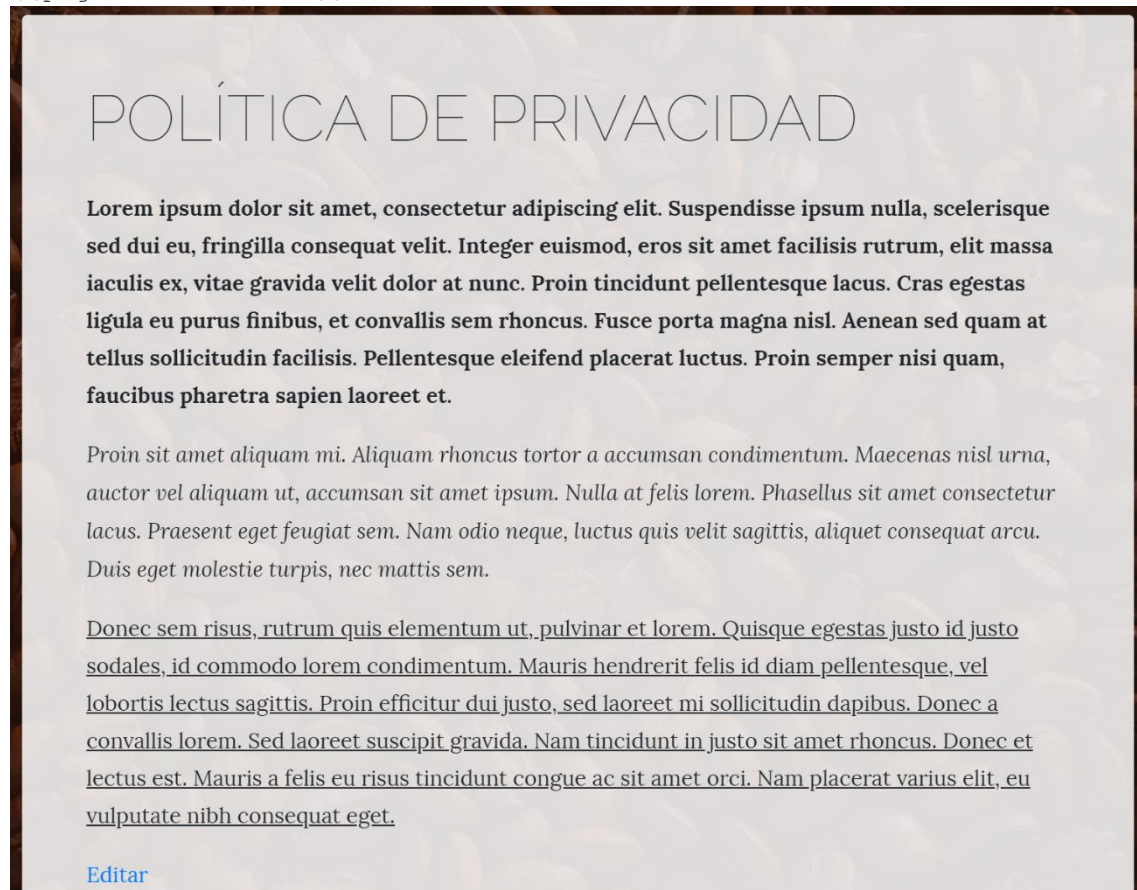


Una vez guardamos el contenido y vamos al frontend, veremos que nos aparece el código HTML no interpretado:



Le tenemos que decir a Django, tranquilo, este contenido contiene HTML seguro así que interprétalo. Eso lo haremos sustituyendo el template tag **linebreaks** por **safe**:

```
pages/templates/pages/sample.html
{{page.content|safe}}
```



Si queréis practicar un poco más podéis poner el editor WYSIWYG en el contenido de las entradas del blog.

Séptima App [Contact] Formularios

Durante las próximas cuatro lecciones nos introduciremos en el mundo de los formularios de Django, obviamente el objetivo será dotar de vida al formulario de contacto de nuestra web:

- En esta primera lección crearemos la respectiva app y trasladaremos la vista y el template contact que tenemos en Core. Aprenderemos a diseñar el formulario, a utilizarlo en la vista y a mostrarlo en su respectivo template.
- En la segunda lección veremos cómo procesar y validar sus campos al enviarlo
- En la tercera lección os mostraré como fusionar el formulario para respetar el diseño del frontend.

- Para acabar esta serie añadiremos la funcionalidad de enviar emails, configuraremos un email real y lo probaremos.

App Contact

Empecemos creando la app contacto:

```
python manage.py startapp contact
```

Trasladamos la vista `contacto` a la nueva app:

```
webempresa/contact/views.py
from django.shortcuts import render

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

Creamos las url de la app:

```
webempresa/contact/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.contact, name="contact"),
]
```

Las añadimos a las url del proyecto:

```
webempresa/webempresa/urls.py
from django.contrib import admin
from django.urls import path, include
from django.conf import settings

urlpatterns = [
    path('', include('core.urls')),
    path('services/', include('services.urls')),
    path('blog/', include('blog.urls')),
    path('page/', include('pages.urls')),
    path('admin/', admin.site.urls),

    # Paths de pages
    path('contact/', include('contact.urls')),
]

if settings.DEBUG:
    from django.conf.urls.static import static
    urlpatterns += static(
        settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Activamos la app:

```
webempresa/webempresa/settings.py
INSTALLED_APPS = [
    'contact',
]
```

Movemos el antiguo template de `core/templates/core/contact.html` a la app `contact/templates/contact/contact.html`

Formulario de contacto

El caso es que cuando necesitemos un formulario, Django nos ofrece la posibilidad de crear un diseño de forma muy parecida a cómo se crean los modelos, de manera que él mismo se encargará de generar el HTML resultante. Además nos provee de una serie de métodos para procesarlo y validar sus campos.

Normalmente se crean en un fichero `forms.py` dentro de la respectiva app, heredando de una clase llamada `Form` que hay en el módulo `forms`. Nosotros vamos a crear el nuestro en la app `Core`, que es la que gestiona la vista `Contact`:

```
webempresa/contact/forms.py
from django import forms

class ContactForm(forms.Form):
    pass
```

Como os he dicho es parecido a crear un modelo, ya que debemos indicar los campos y su tipo. El nuestro tiene tres: un nombre que será una cadena de texto, un email que tiene su propio tipo y el contenido, es lo mínimo necesario para que alguien pueda enviarnos un mensaje y le podamos responder.

```
webempresa/contact/forms.py
from django import forms

class ContactForm(forms.Form):
    name = forms.CharField(
        label="Nombre", required=True)
    email = forms.EmailField(
        label="Email", required=True)
    content = forms.CharField(
        label="Contenido", required=True, widget=forms.Textarea())
```

Como podéis notar sus campos vienen definidos en el módulo `forms` en lugar de `models` y para el nombre se utiliza el atributo `label` en lugar de `verbose_name`. Por defecto estos campos se renderizan como tags `<input>`, pero se pueden cambiar estableciendo un tipo de widget, como en el caso del contenido donde queremos mostrar un tag `<textarea>`.

Hay campos para todo: cadenas, números, emails, fechas, opciones desplegables, ficheros, etc. Os adjunto [un enlace](#) por si queréis aprender más.

Sea como sea ya tenemos diseñado el formulario, así que vamos a utilizarlo, pero antes comentaremos el que tenemos en el template `contact.html`, porque ese es sólo de prueba. Más adelante lo adaptaremos, pero por ahora nos centraremos en lo importante.

```

<!-- Formulario de contacto -->
<!--
<form>...
</form>
-->
<!-- Fin formulario de contacto -->

```

Importamos el formulario, creamos una instancia en la vista y la enviamos al template:

```
from .forms import ContactForm

def contact(request):
    contact_form = ContactForm
    return render(request, 'core/contact.html', {'form': contact_form})

```

Ahora en nuestro template contact.html vamos a dibujar el formulario:

```

<!-- Formulario de contacto -->
<table>
|     {{form.as_p}}
</table>
<!--
<form>...
</form>
-->

```

ENVÍANOS TUS DUDAS

CONTACTO

Nombre:

Email:

Contenido:

¿Habéis visto que fácil es que nos lo dibuje automáticamente?

Por defecto hemos dibujado en formulario como una tabla, de ahí que lo hayamos puesto entre el tag `<table>`, pero también se puede dibujar como párrafos o una lista:

```
{{form.as_p}}
```

```
<ul>  
  {{form.as_ul}}  
</ul>
```


Nombre:

Email:

Contenido:

- Nombre:
- Email:
- Contenido:

Personalmente prefiero la forma de tabla, así que lo dejaré como estaba:

```
<table>
|   {{form.as_table}}
</table>
```

Sin embargo Django no dibuja todo el formulario, sólo los campos. No hay ningún botón para procesarlo, y si inspeccionamos el código veremos también falta el tag `<form>`, necesario para crear formularios HTML:

```

    <table>
      <tr><th><label for="id_name">Name:</label></th><td><input
type="text" name="name" required id="id_name" /></td></tr>
<tr><th><label for="id_email">Email:</label></th><td><input type="email" name="email"
required id="id_email" /></td></tr>
<tr><th><label for="id_content">Content:</label></th><td><textarea name="content"
cols="40" rows="10" required id="id_content">
</textarea></td></tr>
    </table>
```

Seguimos en la segunda parte.

Procesado y validación

Ya tenemos el formulario pero todavía le faltan algunos campos para poder funcionar, vamos a ello.

Lo primero es configurarlo correctamente, así que vamos a envolver un tag `<form>` y a configurar su acción y método:

```
<form action="" method="POST">
  <table>
    |   {{form.as_table}}
  </table>
  <input type="submit" value="Enviar" />
</form>
```

Hay dos métodos para enviar un formulario: POST y GET. El método GET es visible a simple vista, se añade a la URL de la petición con un interrogante al final. A nosotros no nos interesa que las peticiones se vean en la barra de direcciones, por eso vamos a utilizar el método POST que se envía oculto:

En cuanto al atributo `action` sería la página donde enviamos el formulario, al no establecer ningún valor, se interpretará que la petición POST debe realizarse contra la página actual, que en nuestro caso será `/contact/` de la web.

Si intentamos enviar el formulario, como hemos indicado que los campos son obligatorios (con el atributo `required=True`) algunos navegadores nos maneja una prevalidación automática:

Formulario

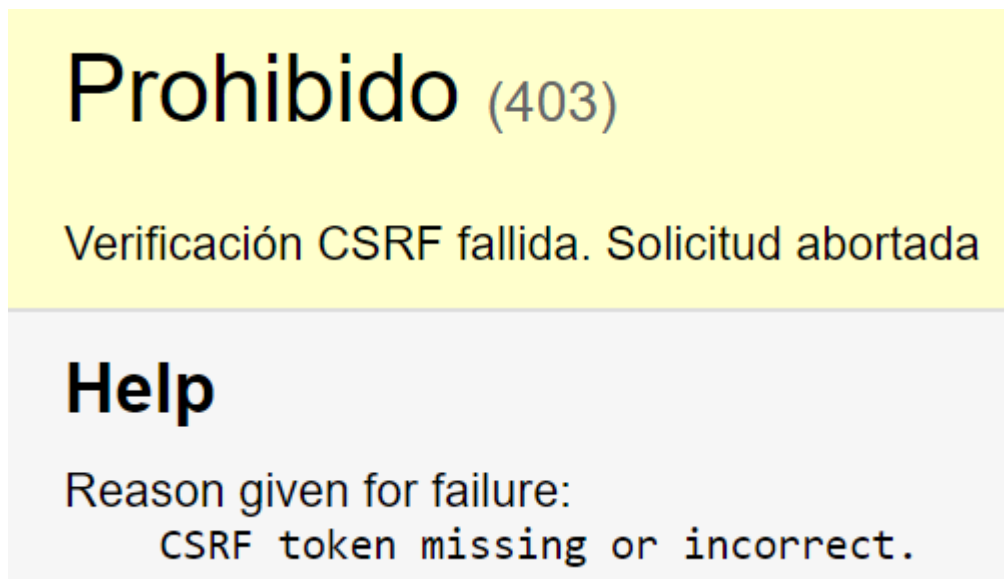
Name:

Email:

Content:

texto de prueba

Cuando consigamos enviar el formulario nos aparecerá un error:



Nos devuelve un error de verificación CSRF fallida (el token CSRF se ha perdido o es incorrecto). ¿Qué es esto del token CSRF? Pues ni más ni menos que un sistema de seguridad. CSRF son las siglas de Cross-site request forgery, en español: Falsificación de petición en sitios cruzados.

¿Para qué sirve? Pues para prevenir que una página web pueda enviar información a nuestros formularios desde un dominio externo. Os adjuntaré en los recursos el enlace a la Wikipedia por si queréis una explicación más detallada. https://es.wikipedia.org/wiki/Cross-site_request_forgery

Para solucionar el error debemos generar un token CSRF dentro del formulario de la siguiente forma:

```
<form action="" method="POST">
    {% csrf_token %}
```

Si recargamos la página a simple vista no habrá cambiado nada, pero si observamos el código veremos un campo hidden (oculto) con el token:

```
<input type='hidden' name='csrfmiddlewaretoken' value='P0BzNwjYP3pPEQddfIxDwF1CurK7M0A1u0lyEeN0EkMIyIacoRv1VMFK9o6hkKIo' />
```

Ahora si volvemos a enviar el formulario, como podréis observar no ocurre nada, pero eso no significa que no se esté enviando, sólo que no lo vemos porque el método es POST. Si cambiamos un momento a GET veréis como se envían en la barra de direcciones:

```
127.0.0.1:8000/contact/?csrfmiddlewaretoken=w9egYUBktxQ5SI...
```

Vamos a dejarlo como estaba:

```
<form action="" method="post">
→  {% csrf_token %}
  <table>
    |   {{form.as_table}}
  </table>
  <button type="submit">Enviar</button>
</form>
```

Para que veais que no os miento, haced lo siguiente:

```
<!-- Formulario de contacto -->
<form action="" method="POST">
  {% csrf_token %}
  <table>
    |   {{form.as_table}}
  </table>
  <input type="submit" value="Enviar" />
</form>
{{request.POST}}
```

Así debuguaremos los datos POST de la petición:

```
Enviar
<QueryDict: {'csrfmiddlewaretoken':
['pLaFkR5dGBxbj7j9kQRDdXJpr1sILpS9EIzuUMZMIQXbW1q3AXV7LC9SQxZBjxy'], 'name': ['test'],
'email': ['sirservorius@gmail.com'], 'content': ['sdssd']}>
```

Queda demostrado entonces que estamos enviando datos a la vista a través de la petición, ahora sólo nos falta manipularlos en nuestra vista para procesarlos y enviar el email.

¿Recordáis que en todas las vistas siempre estamos tomando un parámetro request en la primera posición? Pues, de la misma forma que hemos hecho en el template también podemos acceder a los datos POST.

Sin embargo, antes de lanzarnos a manipular estos datos debemos estar seguros de que ha ocurrido una petición POST. Poned esto en la vista:

```
print("Tipo de petición: ", request.method)
```

Ahora si cargamos nuestra página y observamos la terminal, veremos el tipo de petición que se realiza. Si es la página normal recibimos una petición GET:

```
Tipo de petición: GET
```

En cambio si enviamos el formulario:

```
Tipo de petición: POST
```

Por tanto la respuesta a la pregunta, ¿cuándo procesaremos el formulario? está clara "Sólo cuando detectemos una petición POST":

```
def contact(request):  
    contact_form = ContactForm  
  
    if request.method == "POST":  
        # Procesamos el formulario  
        pass
```

Una vez estamos seguros de que se ha enviado el formulario, lo que haremos es rellenarlo con los datos que se envían. Es fácil, sólo debemos hacer lo siguiente:

```
if request.method == "POST":  
    contact_form = contact_form(data=request.POST)
```

Únicamente habiendo hecho este cambio, si enviamos el formulario, veréis que éste queda rellenado con los campos que se le envían:

Name:	<input type="text" value="Hector Costa"/>
Email:	<input type="text" value="sirservorius@gmail.com"/>
	<input type="text" value="dfdf"/>
Content:	<input type="text"/>

Una utilidad relacionada con esto es que si algún campo no valida bien, por lo menos el usuario no deberá introducir de nuevo todo desde el principio, pero esa no es su razón de ser.

Haber rellenado el formulario nos permite comprobar si todos los campos son válidos:

```
if contact_form.is_valid():  
    pass
```

Si todos los campos son válidos procederemos a recuperarlos. Como `request.POST` no deja de ser un diccionario, una forma segura de hacerlo es utilizar su método `get` que permite devolver un valor por defecto:

```
if contact_form.is_valid():  
    name = request.POST.get('name', '')  
    email = request.POST.get('email', '')  
    content = request.POST.get('content', '')
```

En este punto ya tenemos la información recuperada en la vista, así que procederemos a enviar el email, pero esto lo dejo para más adelante.

Por ahora vamos a suponer que todo ha ido bien y debemos informar al usuario de que su mensaje se ha enviado correctamente. Esto lo podemos hacer fácilmente. En lugar de renderizar el template de contacto podemos hacer un redireccionamiento enviando una variable `OK` por GET a la propia página `/contact/`.

Para redireccionar tenemos a nuestra disposición la función `redirect`:

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

Y podríamos simplemente hacer lo siguiente:

```
# Suponemos que todo ha funcionado, redireccionamos
return redirect('/contact/?ok')
```

Esto está bien, pero ya sabéis que no me gusta poner cadenas en crudo, es una mala práctica. Por ello voy a introducir brevemente la función `reverse()`, que es igual que `{% url %}` en los templates:

```
from django.urls import reverse
\
return redirect(reverse('contact') + "?ok")
```

Hacedme caso, parece tedioso pero es una muy buena práctica dejar que Django resuelva él mismo las URL que tenemos definidas en los Paths.

Ahora que tenemos la redirección, vamos a enviar el formulario y como véis nos recarga la página pasando este OK en la parte superior.

127.0.0.1:8000/contact/?ok

Ahora en el template podemos comprobar si existe esta variable GET 'ok' en el diccionario `request.GET`, y si es así mostrar un mensaje que diga "Su mensaje se ha enviado correctamente etc.":

```
<!-- Formulario de contacto -->
{% if 'ok' in request.GET %}
    <p><b>Su mensaje se ha enviado correctamente, en breve nos pondremos en contacto.</b></p>
{% endif %}
<form action="" method="POST">
```

\

Su mensaje se ha enviado correctamente, en breve nos pondremos en contacto.

El formulario está casi listo, pero visualmente no tiene nada que ver con el que teníamos maquetado. En la siguiente lección os enseñaré como adaptar un diseño en lugar de utilizar el que genera Django automáticamente.

Fusionando el formulario

Lo que vamos a hacer es reutilizar nuestra maqueta y cambiar sólo las partes dinámicas. Para ello dibujaremos los campos manualmente en lugar de hacer que Django lo muestre todo de golpe. Este proceso viene detallado en la documentación, como siempre os comparto el enlace en los recursos.

<https://docs.djangoproject.com/en/dev/topics/forms/#rendering-fields-manually>

Vamos a sustituir la parte del renderizado por nuestro código:

```
<!-- Formulario de contacto -->
{% if 'ok' in request.GET %}
    <p>Su mensaje se ha enviado correctamente, en breve nos pondremos en contacto.</p>
{% endif %}
<form action="" method="POST">
    {% csrf_token %}
    <div class="form-group">
        <label>Nombre *</label>
        <div class="input-group">
            <input type="text" class="form-control">
        </div>
    </div>
    <div class="form-group">
        <label>Email *</label>
        <div class="input-group">
            <input type="text" class="form-control">
        </div>
        <!--<ul class="errorlist">
            <li>El email no es correcto.</li>
        </ul>-->
    </div>
    <div class="form-group">
        <label>Mensaje *</label>
        <div class="input-group">
            <textarea class="form-control"></textarea>
        </div>
    </div>
    <div class="text-center">
        <input type="submit" class="btn btn-primary btn-block py-2" value="Enviar">
    </div>
    <input type="submit" value="Enviar" />
</form>
<!-- Fin formulario de contacto -->
```

Ahora vamos a sustituir los campos Nombre, Email y Mensaje por lo siguiente:

```

<!-- Formulario de contacto -->
{% if 'ok' in request.GET %}
    <p>Su mensaje se ha enviado correctamente, en breve nos pondremos en contacto.</p>
{% endif %}
<form action="" method="POST">
    {% csrf_token %}
    <div class="form-group">
        <label>Nombre *</label>
        <div class="input-group">{{form.name}}</div>
    </div>
    <div class="form-group">
        <label>Email *</label>
        <div class="input-group">{{form.email}}</div>
        <!--<ul class="errorlist">
            <li>El email no es correcto.</li>
        </ul>-->
    </div>
    <div class="form-group">
        <label>Mensaje *</label>
        <div class="input-group">{{form.content}}</div>
    </div>
    <div class="text-center">
        <input type="submit" class="btn btn-primary btn-block py-2" value="Enviar">
    </div>
    <input type="submit" value="Enviar" />
</form>
<!-- Fin formulario de contacto -->

```

Obviamente hacerlo de esta forma es más trabajoso pero también más elegante, ya que podemos añadir estilos a la validación de errores. Justo después de cada campo podemos mostrar sus errores específicos con `{{ form.campo.errors }}`. Si existen errores Django genera una lista con la clase 'errorList' (ver código fuente). Como yo ya he maquetado de antemano los estilos de esa clase nos saldrá bien:

```

<div class="form-group">
  <label>Nombre *</label>
  <div class="input-group">{{form.name}}</div>
  {{form.name.errors}}
</div>
<div class="form-group">
  <label>Email *</label>
  <div class="input-group">{{form.email}}</div>
  {{form.email.errors}}
</div>
<div class="form-group">
  <label>Mensaje *</label>
  <div class="input-group">{{form.content}}</div>
  {{form.content.errors}}
</div>

```

Vamos a ver como queda:

Nombre *

El nombre no puede estar vacío

Email *

El email no es válido

Mensaje *

El mensaje no puede estar vacío

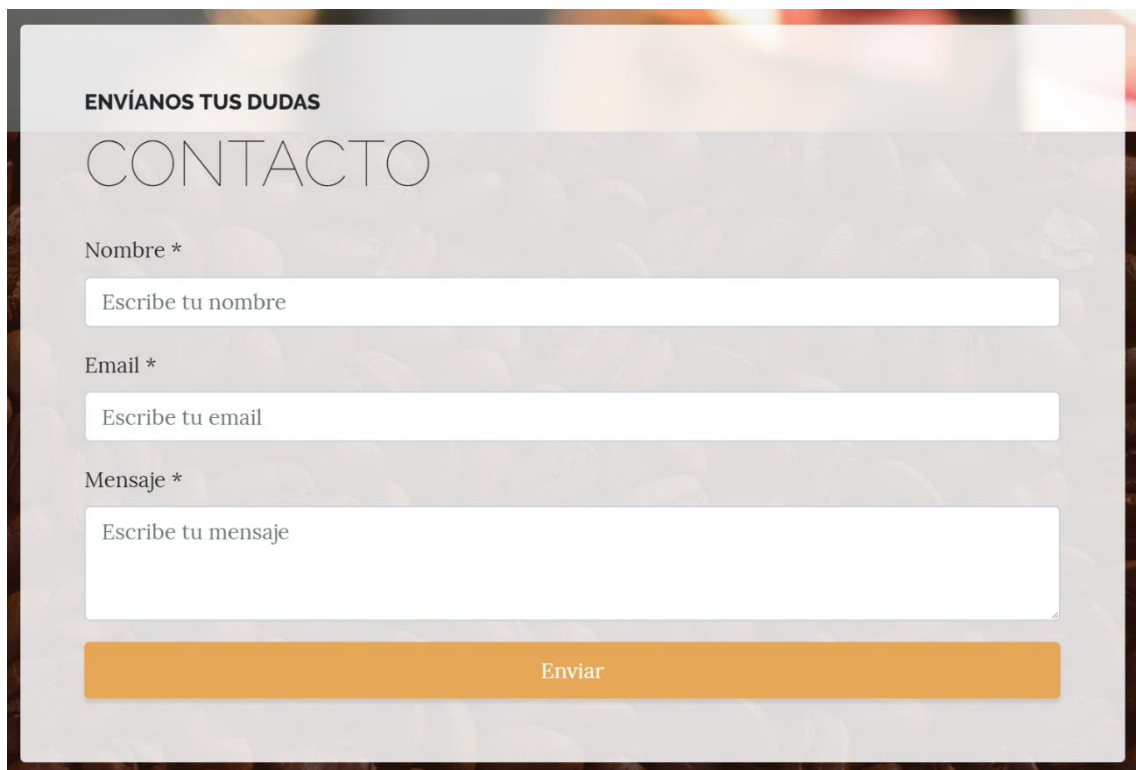
Enviar

Va tomando forma, pero todavía nos faltan unos pequeños detalles: los atributos internos de los inputs y el textarea.

Estas configuraciones nos permitirán modificar las longitudes mínimas y máximas, o cualquier atributo del campo sobreescribiendo el diccionario attrs a partir del widget, lo cual es útil para añadir una clase css al campo y darle la apariencia que tenía en nuestra maqueta, entre otras muchas cosas:

```
class ContactForm(forms.Form):
    name = forms.CharField(label='Nombre', min_length=3, max_length=100, required=True,
        widget=forms.TextInput(attrs={'placeholder': 'Escribe tu nombre', 'class': 'form-control'}))
    email = forms.EmailField(label='Email', min_length=3, max_length=100, required=True,
        widget=forms.TextInput(attrs={'placeholder': 'Escribe tu email', 'class': 'form-control'}))
    content = forms.CharField(label='Mensaje', min_length=10, max_length=1000, required=True,
        widget=forms.Textarea(attrs={'placeholder': 'Escribe tu mensaje', 'rows': 3, 'class': 'form-control'}))
```

Y con esto ahora sí tenemos el formulario perfecto, sólo nos faltará implementar el envío de emails:



Enviando emails

Django incluye un módulo dedicado expresamente a manipular y enviar emails. Dentro encontramos funciones como `send_message` para usos simples o la clase `EmailMessage`, más sofisticada. Nosotros utilizaremos la segunda forma, ya que nos permite establecer fácilmente un `reply_email`, o email de respuesta, ya lo veréis.

Antes de nada para enviar emails necesitamos configurar una cuenta de correo.

Podríamos estar horas y horas hablando sobre como configurar diferentes correos con sus diferentes protocolos pero eso sería un lío.

Lo que os recomiendo es registrarnos en un proveedor como mailtrap, ya que nos permite crear cuentas de correo gratuitas para realizar pruebas, y además funciona en Django sin problemas. Luego en la práctica cada uno deberá configurar su cuenta, os dejaré la documentación:

<https://docs.djangoproject.com/en/dev/topics/email/>

Una vez establecida la configuración de una forma segura estamos listos para ir a la vista y enviar un nuestro correo de contacto utilizando la función `send_mail`.

Primero importaremos la función y el EMAIL que utilizaremos para enviar el correo, tomado del propio `settings.py`:

```
from django.core.mail import send_mail
from webempresa.settings import EMAIL_HOST_USER
```

Luego creamos la estructura del correo utilizando la clase `EmailMessage` (os adjunto la documentación relacionada): <https://docs.djangoproject.com/en/dev/topics/email/#emailmessage-objects>

DONDE `EMAIL_HOST_USER` -> EMAIL DEL QUE ENVIA

```
# Enviamos el correo y redireccionamos
email = EmailMessage(
    "La Caffetiera: Nuevo mensaje de contacto",          # Asunto
    "De {} <{}>\n\nEscribió:\n\n{}".format(name, email, content), # Cuerpo
    EMAIL_HOST_USER,                                     # Email de origen
    ['redacted@redacted.com'],                           # Email de destino
    reply_to=[email]                                     # Email de respuesta
)
try:
    email.send()
    # Todo ha ido bien, redireccionamos a OK
    return redirect(reverse('contact') + "?ok")
except:
    # Algo no ha ido bien, redireccionamos a FAIL
    return redirect(reverse('contact') + "?fail")
```

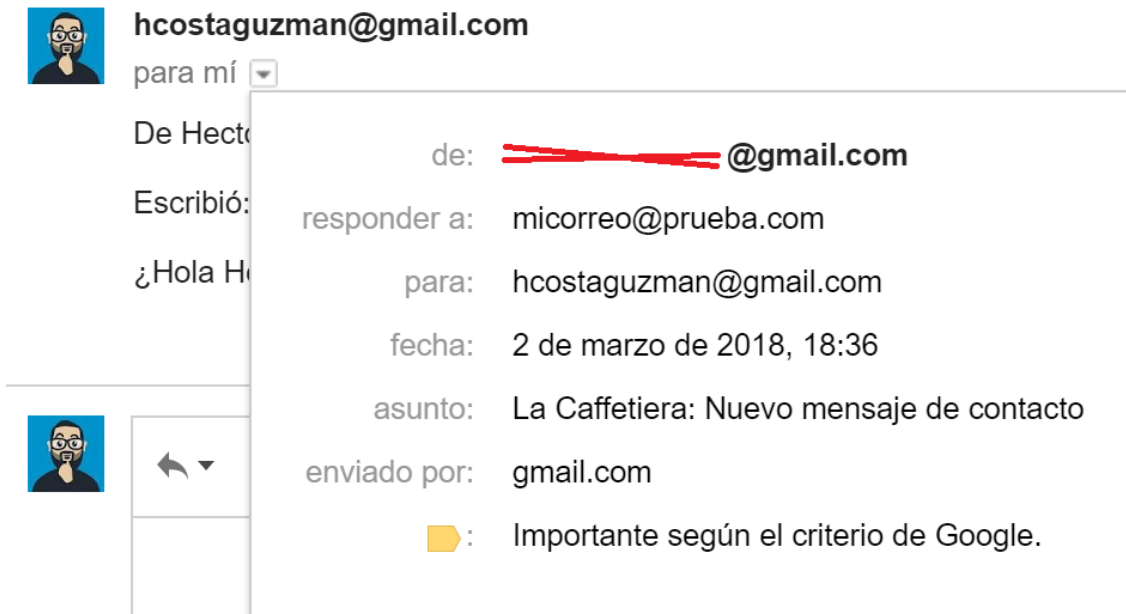
Cuando la tenemos, utilizamos el método `send()` para enviarlo. Podemos detectar algún fallo con un bloque `try except` y redireccionar tanto si funciona bien como si falla:

```
try:
    email.send()
    # Todo ha ido bien, redireccionamos a OK
    return redirect(reverse('contact') + "?ok")
except:
    # Algo no ha ido bien, redireccionamos a FAIL
    return redirect(reverse('contact') + "?fail")
```

Sólo deberíamos modificar el template y detectar si en request.GET está el parámetro FAIL para mostrar un mensaje clásico de “Error enviando el formulario, prueba más tarde”, pero eso os lo dejo a vosotros.

Vamos a probar si se envía el correo:

The image shows a web application interface for a contact form. The form is titled "ENVÍANOS TUS DUDAS" and "CONTACTO". It contains three input fields: "Nombre *" with the value "Hector Costa", "Email *" with the value "micorreo@prueba.com", and "Mensaje *" with the value "¿Hola Héctor qué la web empresarial?". Below the fields is an orange "Enviar" button. The background of the form is a blurred image of coffee beans. Below the form, there is a browser window showing a dark-themed interface. The browser's address bar shows "yo" and the page title is "La Caffetiera: Nuevo mensaje de contacto".

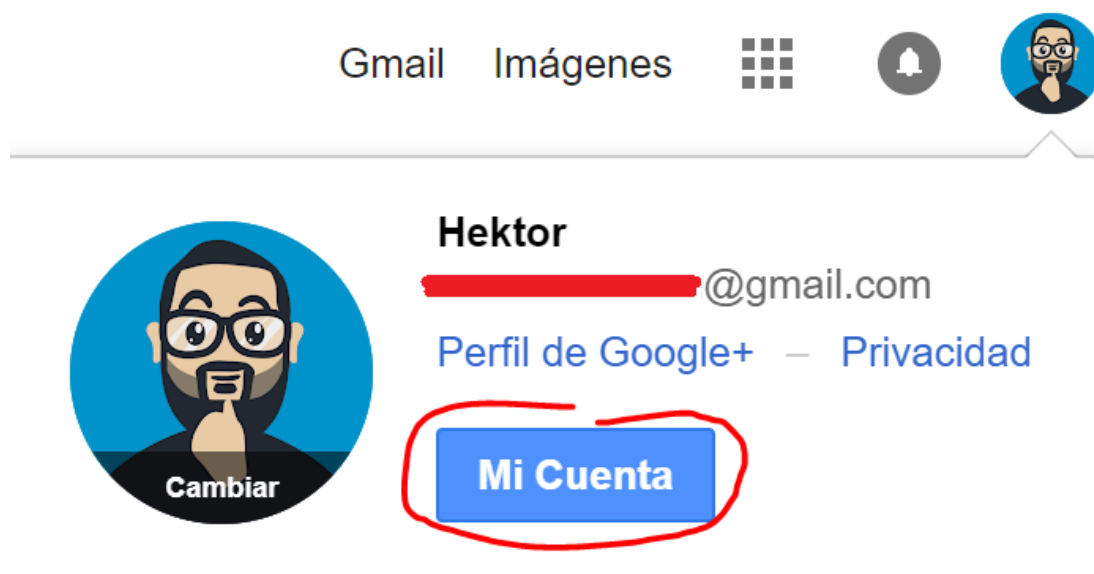


Y bien, por lo menos a mi me ha funcionado sin problemas.

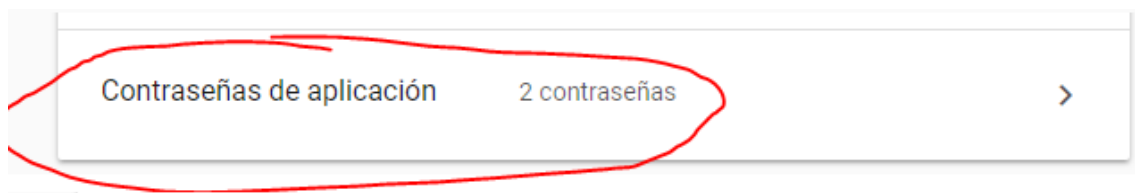
Evidentemente en la vida real no se recomienda utilizar un email de Google, sobretodo si vamos a enviar muchos correos, pero para una web sencilla puede servir perfectamente.

Con esto tenemos el proyecto prácticamente terminado, sólo faltan algunos ajustes en el admin para prevenir ciertas acciones por parte de nuestros clientes, os lo cuento en detalle en la próxima lección.

Así que, suponiendo que ya tenéis una cuenta de Gmail, aunque sea de prueba, vamos a empezar generando una clave de aplicación. Esto es necesario porque por seguridad Gmail no nos deja identificarnos con nuestra en aplicaciones externas, necesitamos un token. El primer paso es ir a nuestra cuenta de Google desde cualquier web de Google:



Vamos al apartado Inicio de sesión y seguridad > Contraseñas de aplicación



Aquí generaremos una contraseña para Correo, podemos darle el nombre Django Test
Email:\

Correo



Seleccionar dispositivo



Selecciona la aplicación y el dispositivo para los que quieres generar la contraseña de aplicación.

Correo



Seleccionar dispositivo

iPhone

iPad

BlackBerry

Mac

Windows Phone

Ordenador con Windows

Otra (nombre personalizado)

GENERAR

Una vez la tengamos la dejáis reservada, luego la utilizaremos.

Contraseña de aplicación generada

Tu contraseña de aplicación para el dispositivo

kohs ~~XXXXXXXXXXXX~~

El siguiente paso es configurar el email en el fichero settings.py. Debéis poner exactamente esto:


```
# Email config
EMAIL_HOST = ''
EMAIL_HOST_USER = ''
EMAIL_HOST_PASSWORD = ''
EMAIL_PORT = ''
EMAIL_USE_TLS = ''
DEFAULT_FROM_EMAIL = EMAIL_HOST_USER
```

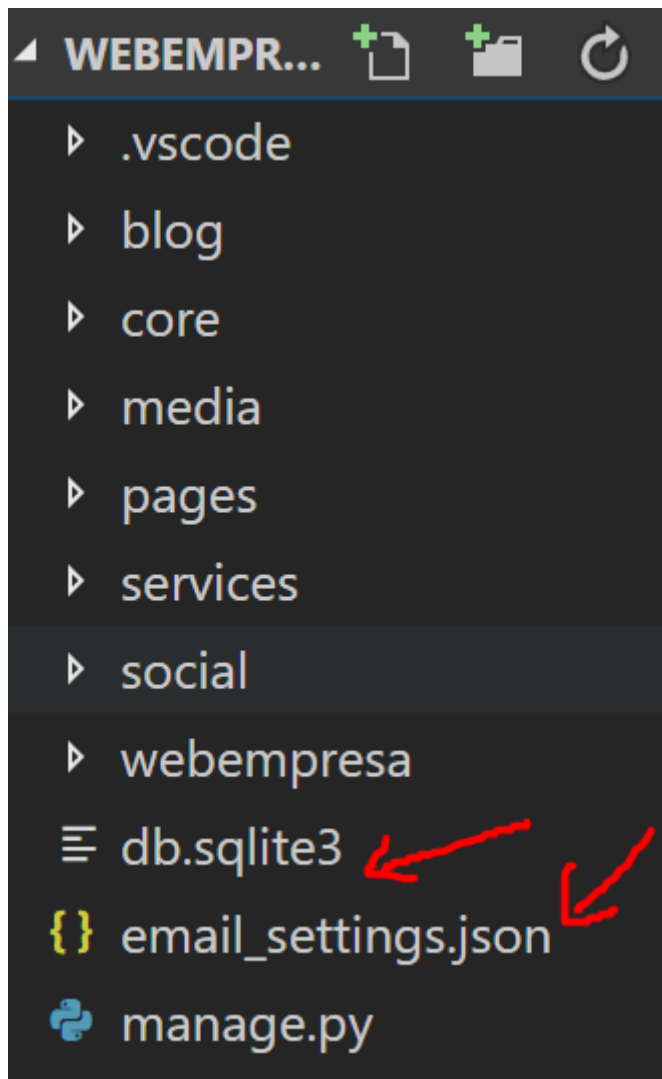
Ahora viene cuando la matan, tenemos que configurar los campos.

El problema que tiene esto, es que no es el tipo de información que podemos poner en cualquier sitio.

Algunos desarrolladores cuando tienen que utilizar información confidencial optan por crear lo que se conoce como variables de entorno. Es una práctica válida y recomendada, pero no vamos a utilizarla en este curso. En su lugar os voy a enseñar a crear un fichero de configuración externo y a cargarlo en Django para hacer uso de sus datos.

Vamos a crear un fichero de configuración JSON, un formato de serialización de datos bastante común hoy en día. Nosotros lo vamos a crear en el directorio raíz del proyecto, pero si lo váis a utilizar en producción os sugiero ponerlo en otro directorio del equipo, y sobretodo, no publicarlo nunca en un repositorio público.

A este fichero lo llamaremos `email_settings.json`:



Dentro escribiremos la siguiente estructura, muy similar a un diccionario de Python (atención a true y no True):

A screenshot of a code editor showing the content of a file named 'email_conf.json'. The file contains a JSON object with the following key-value pairs: 'EMAIL_HOST' with value 'smtp.gmail.com', 'EMAIL_HOST_PASSWORD' with a redacted password, 'EMAIL_HOST_USER' with a redacted email address followed by '@gmail.com', 'EMAIL_PORT' with value 587, and 'EMAIL_USE_TLS' with value true. The file name 'email_conf.json' is underlined with a red line in the editor's tab bar.

```
{  
  "EMAIL_HOST": "smtp.gmail.com",  
  "EMAIL_HOST_PASSWORD": "REDACTED",  
  "EMAIL_HOST_USER": "REDACTED@gmail.com",  
  "EMAIL_PORT": 587,  
  "EMAIL_USE_TLS": true  
}
```

Una vez lo tengamos vamos a settings.py y lo cargaremos utilizando el módulo json de python de la siguiente forma:

```
import json
```

```
\n# Email config: Must create email_settings.json in project root\nEMAIL_SETTINGS_FILE = os.path.join(BASE_DIR, 'email_settings.json')\nwith open(EMAIL_SETTINGS_FILE) as data_file:\n    email_settings = json.load(data_file)\n\nEMAIL_HOST = email_settings['EMAIL_HOST']\nEMAIL_HOST_USER = email_settings['EMAIL_HOST_USER']\nEMAIL_HOST_PASSWORD = email_settings['EMAIL_HOST_PASSWORD']\nEMAIL_PORT = email_settings['EMAIL_PORT']\nEMAIL_USE_TLS = email_settings['EMAIL_USE_TLS']
```

Personalizando el administrador (4)

Nuestra web está acabada, pero debemos tener en cuenta un detalle importante, y es que nuestros clientes no deberían tener permisos para gestionar algunas partes del panel de administrador. Puede sonar cruel, pero hay que pensar en el cliente como alguien... vamos a decir despistado, al que si le decimos que no haga algo seguro que lo acaba haciendo, os lo digo con total seguridad.

Por tanto como dice el dicho, ojos que no ven corazón que no siente, y por tanto más que advertirle lo que haremos es esconder las partes peligrosas. Para lograrlo vamos a introducir el uso de los grupos y permisos de usuario.

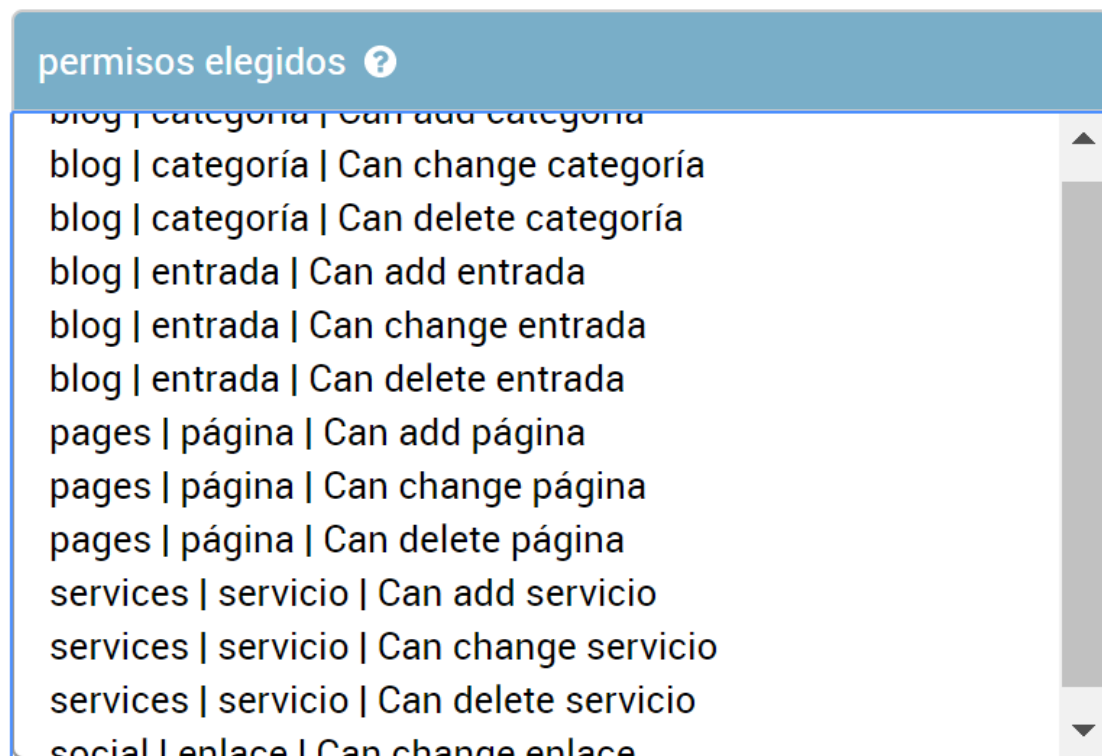
Vamos a ir a nuestro panel de administrador con nuestro superusuario y vamos a ir al apartado grupos. Crearemos un nuevo grupo llamado "personal". Supondremos que es para el personal de la empresa.

Los permisos se gestionan de la siguiente forma. Para cada modelo hay tres permisos: añadir, editar y borrar. Es tan sencillo como seleccionar los permisos que queremos dar a los miembros del grupo y ponerlos en el lado derecho, podemos hacer doble clic o seleccionar varios con Shift o Control y presionar el botón de la flecha hacia la derecha.

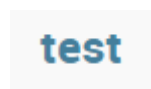
Para los miembros del grupo Personal vamos a añadir los siguientes permisos:

- App Blog: Todos los permisos para los modelos Categoría y Entrada
- App Pages: Todos los permisos para el modelo Página
- App Services: Todos los permisos para el modelo Servicio
- App Social: Permiso de edición para el modelo Enlace

En resumen, dejaremos que el usuario maneje completamente el blog y las páginas secundarias, pero los servicios y los enlaces sociales los restringimos para que no pueda añadir de nuevos ni borrar los actuales:



Ahora tenemos que crear un usuario y añadirlo a este grupo. Podemos llamarlo test:



Deberemos marcarlo como STAFF para que tenga acceso al admin, y añadirlo al grupo de Personal:

☒ Es staff
Indica si el usuario puede entrar en este sitio de administración.

☐ Es superusuario
Indica que este usuario tiene todos los permisos sin asignárselos explícitamente.

Grupos:



grupos Disponibles ?
Q Filtro


grupos elegidos ?
Personal


Como podéis observar se pueden configurar varios campos y permisos específicos para el usuario, yo os recomiendo gestionar un grupo porque es más cómodo.

Vamos a entrar con este usuario test a ver qué nos aparece:\

Sitio administrativo

BLOG		
Categorías	+ Añadir	 Modificar
Entradas	+ Añadir	 Modificar

GESTOR DE PÁGINAS		
Páginas		 Modificar

REDES SOCIALES		
Enlaces		 Modificar

Todo lo tenemos perfecto excepto por una cosa.

En la App Social, estamos permitiendo que el usuario pueda cambiar la CLAVE y el NOMBRE de la red. Dentro de lo que cabe el nombre no importa mucho, pero la CLAVE la utilizamos directamente en el template, así que no podemos dejarle editarla.

Los permisos nos pueden ayudar a nivel de Modelo, pero no de campos, así que tendremos que encontrar una forma de por lo menos hacer el campo “sólo de lectura” a los miembros del grupo “Personal”.

¿Recordáis cómo hicimos que un campo fuera sólo de lectura? Teníamos que editar crear una tupla o lista llamada `readonly_fields` en el admin del modelo. Claro, esto no podemos hacerlo porque es común para todos los usuarios, pero hay una forma de extenderlo: crear un método `get_readonly_fields` capaz de detectar el usuario identificado durante la petición y sobrescribir su valor:

```
class LinkAdmin(admin.ModelAdmin):
    def get_readonly_fields(self, request, obj=None):
        if request.user.groups.filter(name='Personal').exists():
            return ('key', 'name')
        else:
            return ()
```

De esta manera, si detectamos que el usuario actual forma parte del grupo Personal, le cambiamos dinámicamente los campos `key` y `name` como sólo lectura. Y si no devolvemos una tupla vacía indicando que puede editarlos todos (demo).

La mayoría de campos del ModelAdmin se pueden sobrescribir en tiempo de ejecución con sus respectivos métodos, yo suelo utilizar mucho el `get_exclude`, que permite esconder campos en lugar de ponerlos sólo de lectura. Os dejaré un enlace a la documentación por si queréis investigar.

<https://docs.djangoproject.com/en/dev/ref/contrib/admin/#modeladmin-methods>

Y ya está, sólo quiero daros la enhorabuena por haber completado el segundo proyecto del curso. Deseo de corazón que hayáis aprendido mucho y podáis ponerlo todo en práctica en vuestros proyectos.

Nos vemos en el próximo.