

# Proyecto Web Playground

Hasta ahora os he enseñado cómo hacer un par de webs relativamente comunes mientras estábamos aprendiendo, pero ha llegado el momento de meter las manos en el código. Para este tercer y último proyecto del curso, en lugar de una web como las dos anteriores, os propongo crear un playground. ¿Qué qué es un playground? Pues un lugar para aprender experimentando.

En definitiva vamos a meternos de lleno en Django, aprenderemos un montón de conceptos avanzados y estaremos escribiémos mucho código. Poneos cómodos y preparad los asientos porque empezamos con la "Web Playground".

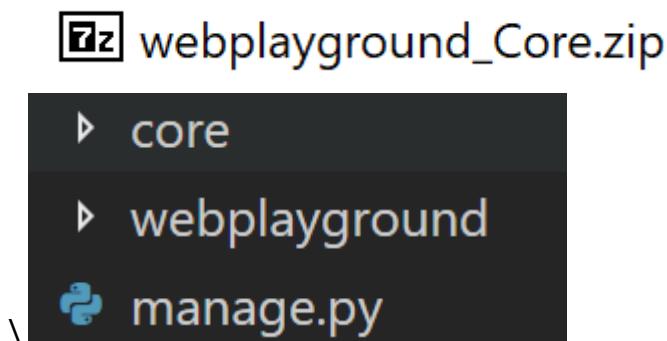
Repositorio: <https://github.com/hektorprofe/webplayground>

## Preparando la App (Core)

Empezaremos creando un proyecto en nuestro entorno django2:

```
› django-admin startproject webplayground
```

Ahora os voy a proporcionar una app Core preparada para este proyecto, es en esencia igual que la de la webempresa, pero en ella he adaptado los templates a algo más genérico. Por defecto trae dos páginas: home y una genérica, sample:



Sólo tenemos que añadirla a nuestra lista de APPS en settings y en el urls.py del proyecto:

```
'core',  
\  
from django.contrib import admin  
from django.urls import path, include  
  
urlpatterns = [  
    # Paths del core  
    path('', include('core.urls')),  
    # Paths del admin  
    path('admin/', admin.site.urls),  
]  
]
```

Comprobamos si funciona:



Bien, ahora ya podemos empezar.

## Class Based Views y TemplateView

Desde el principio del curso todas las vistas que hemos creado en nuestros proyectos han sido vistas basadas en funciones, FBV. Su nombre está claro, pues se definen en simples funciones que luego llamamos en el respectivo fichero urls.py. Estas eran las únicas vistas que se podían crear hasta que llegaron las vistas basadas en clases.

¿Qué diferencia hay? Bueno, si lo pensáis detenidamente una clase es una estructura mucho más completa que una simple función. Permite usarla de molde y puede contener comportamientos predefinidos que se pueden heredar en otras clases, así que tienen un objetivo muy claro: ahorrarnos tiempos a la hora de implementar funcionalidades comunes.

En otras palabras, tenemos a nuestra disposición un amplio catálogo de Vistas Basadas en Clases a modo de plantillas que podemos utilizar y extender a voluntad.

Hay una web increíble llamada <https://ccbv.co.uk/> donde podemos encontrar un resumen y documentación de todas ellas (entrar y comentarlas):

## Classy Class-Based Views.

Detailed descriptions, with full methods and attributes, for each of Django's class-based generic views.

### Start Here for Django 2.0.

Show more

#### AUTH VIEWS

[LoginView](#)  
[LogoutView](#)  
[PasswordChangeDoneView](#)  
[PasswordChangeView](#)  
[PasswordResetCompleteView](#)  
[PasswordResetConfirmView](#)

#### GENERIC DETAIL

[DetailView](#)

#### GENERIC EDIT

[CreateView](#)  
[DeleteView](#)

### What are class

Django's class-based generic views are common web development tools based on Python's object orientation and its emphasis on code reuse. This means they're more than just views; they provide utilities which can be used to build your own views yourself.

### Great! So what?

All of this power comes at the cost of a bit of extra complexity.

A lo largo de esta web playground haremos uso de un montón de ellas, ya veréis.

Vamos a empezar con la app Core, ¿cómo se traduciría lo que tenemos ahora a CBV? Bueno, lo primero es elegir qué tipo de CBV se adaptaría mejor a nuestras necesidades.

Como las vistas de Blog no tienen nada de especial y simplemente cargan un Template, deberíamos buscar una CBV genérica, concretamente la de `TemplateView`.

Entramos a la documentación: <https://ccbv.co.uk/projects/Django/2.0/django.views.generic.base/TemplateView/> Y de ahí a la documentación de Django: <https://docs.djangoproject.com/en/dev/ref/class-based-views/>

En la documentación encontraremos cómo manejar las `TemplateView`:

Example views.py:

```
from django.views.generic.base import TemplateView

from articles.models import Article

class HomePageView(TemplateView):

    template_name = "home.html"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['latest_articles'] = Article.objects.all()[:5]
        return context
```

Example urls.py:

```
from django.urls import path

from myapp.views import HomePageView

urlpatterns = [
    path('', HomePageView.as_view(), name='home'),
]
```

En este ejemplo nos muestran como crear una CBV TemplateView, pasar valores al diccionario de contexto y cómo llamarla en urls. Nosotros no tenemos que enviar nada, así que nos será más fácil:

```

from django.shortcuts import render, HttpResponseRedirect
from django.views.generic.base import TemplateView

class HomePageView(TemplateView):
    template_name = "core/home.html"

class SamplePageView(TemplateView):
    template_name = "core/sample.html"
\

from django.urls import path
from .views import HomePageView, SamplePageView

urlpatterns = [
    path('', HomePageView.as_view(), name="home"),
    path('sample/', SamplePageView.as_view(), name="sample"),
]

```

Como véis ya no estamos creando funciones que devuelven el template con la lógica, sino clases que heredan de `TemplateView` y que por defecto tienen un atributo llamado `template_name`. Luego a la hora de llamarlas en el `urls` tienen un método llamado `as_view()` que las devuelve como una vista. Vamos a probar:



## Web Playground

CBV, autenticación, registros, perfiles y más

Cómo veis está todo igual. ¿Por qué no probamos a enviar algún dato al diccionario de contexto? Para ello sobreescriviremos el método `get_context_data` que debe devolver un diccionario de contexto después de recuperarlo de la superclase:

```
class HomePageView(TemplateView):
    template_name = "core/home.html"

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['title'] = "Mi super Web Playground"
        return context

\<h1>{{title}}</h1>
<p class="lead">CBV, autenticación, registros, perfiles y más</p>
\
```

# Mi super Web Playground

CBV, autenticación, registros, perfiles y más

Pero como esto es un poco complejo, en lugar de modificar el contexto es posible sobreescribir el método `get` que es el encargado de devolver la respuesta:

```
class HomePageView(TemplateView):
    template_name = "core/home.html"

    def get(self, request, *args, **kwargs):
        return render(request, self.template_name, {'title': 'Mi super Web Playground'})
\
```

# Mi super Web Playground

CBV, autenticación, registros, perfiles y más

¿Esto ya se parece más a una FBV no?

Por cierto, ¿se os ocurre alguna razón por la que este método se llame `get`? Pues sí, se llama GET ni más ni menos porque se encarga de responder las peticiones GET, así que podéis suponer que también existe un método POST, algo muy conveniente para separar la lógica al procesar formularios, pero eso lo veremos más adelante.

Por ahora es posible que muchos os preguntéis para qué utilizar las CBV y complicarnos la vida. Es verdad que cambiar la forma de pensar es difícil, pero creedme que vale la pena y os lo demostraré en las siguientes lecciones.

# Preparando la App [Pages]

Muy bien! Para seguir aprendiendo más sobre las Vistas basadas en Clases he preparado también una copia de nuestra app Pages para hacerle una revisión. Descargadla en los recursos y descomprimidla en vuestro proyecto.

Para utilizarla simplemente la activaremos en settings.py, configuraremos las urls.py y a migraremos la base de datos sin olvidar crear un superusuario para poder acceder al panel de administrador (desde nuestro entorno virtual django2).

```
'pages.apps.PagesConfig',
\

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

\python manage.py migrate
\

python manage.py makemigrations pages
\python manage.py migrate pages
\python manage.py createsuperuser
```

Ahora para acceder a nuestra app vamos a modificar el enlace a la página de prueba por el de esta app pages, simplemente podríamos cambiar el enlace a /páginas/ en el menú, ahí tengo una vista que muestra nuestras páginas y un pequeño extracto de su contenido (borrar la de ejemplo y dejar esta):

```
<ul class="navbar-nav mr-auto">
    <li class="nav-item">
        <a class="nav-link" href="{% url 'home' %}">Inicio</a>
    </li>
    <li class="nav-item">
        <a class="nav-link" href="{% url 'pages' %}">Páginas</a>
    </li>
</ul>
```

The screenshot shows a dark-themed navigation bar with three items: "Playground", "Inicio", and "Páginas". The "Playground" item is highlighted with a red background and white text. The "Inicio" item is highlighted with a blue background and white text. The "Páginas" item is highlighted with a green background and white text. This visual cue indicates which part of the menu structure has been modified or is being demonstrated.

Por cierto, si os preguntáis porque he creado esta vista es para trabajar un tipo concreto de CBV, luego cuando la adaptemos lo veréis. En cualquier caso estoy devolviendo un error 404 si no hay ninguna página, por lo que nos dará error si intentamos

```
acceder:\ pages = get_list_or_404(Page)\
```

## Page not found (404)

**Request Method:** GET

**Request URL:** http://127.0.0.1:8000/pages/

**Raised by:** pages.views.pages

No Page matches the given query.

Vamos a crear algunas páginas de prueba:

GESTOR DE PÁGINAS

Páginas	Añadir	Modificar
---------	--------	-----------

Fijaros que al intentar crear una página nos da error de ckeditor, claro, aunque la tengamos instalada en el entorno virtual no la hemos instalado en settings.py:

TemplateDoesNotExist at /admin/pages/page/add/  
ckeditor/widget.html

```
'ckeditor', ←  
'core',  
'pages.apps.PagesConfig',\
```

Ahora sí ya debería funcionar la App:

## Añadir página

**Etiam pharetra risus eu venenatis sodales**

Etiam pharetra risus eu venenatis sodales. Donec sit amet rhoncus odio, quis malesuada tellus. Aenean non luctus metus. Proin rutrum leo vitae tempor egestas. Phasellus mi tellus, feugiat et orci e...

[Leer más](#)

## **Lorem ipsum dolor sit amet**

*Maecenas ac ante neque. Quisque sed ultricies sapien, a accumsan elit. Sed non vive...*

[Leer más](#)

## Modificar página

Titulo:	Etiā pharetra risus eu venenatis sodales
Contenido:	<p>Etiā pharetra risus eu venenatis sodales. Donec sit amet rhoncus odio, quis malesuada tellus. Aenean non luctus metus. Proin rutrum leo vitae tempor egestas. Phasellus mi tellus, feugiat et orci et, pellentesque elementum augue. Duis tincidunt commodo mauris sit amet posuere. Sed at suscipit est. Vivamus non mi purus. Sed vel velit in ligula aliquam sodales ac id dui. Nulla sit amet nisl velit. Proin eleifend, tellus nec fringilla vulputate, sem dolor bibendum felis, nec luctus augue risus et massa. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;</p>

**Etiam pharetra risus eu venenatis sodales**

Etiam pharetra risus eu venenatis sodales. Donec sit amet rhoncus odio, quis malesuada tellus. Aenean non luctus metus. Proin rutrum leo vitae tempor egestas. Phasellus mi tellus, feugiat et orci et, pellentesque elementum augue. Duis tincidunt commodo mauris sit amet posuere. Sed at suscipit est. Vivamus non mi purus. Sed vel velit in ligula aliquam sodales ac id dui. Nulla sit amet nisl velit. Proin eleifend, tellus nec fringilla vulputate, sem dolor bibendum felis, nec luctus augue risus et massa. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;

[Editar](#)

Con esto tenemos el terreno preparado y en la próxima lección aprenderemos cómo utilizar dos nuevas CBV.

# DetailView y ListView

Ahora mismo nuestra app Pages tiene dos vistas, una para mostrar un resumen de las páginas (pages) y otra para cada página individual (page).

```
def pages(request):  
    pages = get_list_or_404(Page)  
    return render(request, 'pages/pages.html', {'pages':pages})  
  
def page(request, page_id, page_slug):  
    page = get_object_or_404(Page, id=page_id)  
    return render(request, 'pages/page.html', {'page':page})
```

Pues os gustará saber que existen dos CBV pensadas para ambas situaciones, ya sea devolver una instancia o una lista, y son ni más ni menos que `DetailView` y `ListView`, vamos a utilizarlas.

Para implementarlas vamos a inspirarnos en la documentación oficial, a la cual llegamos fácilmente desde <https://ccbv.co.uk/>.

Empecemos por la lista de páginas:

\ <https://ccbv.co.uk/projects/Django/2.0/django.views.generic.list/ListView/> \  
<https://docs.djangoproject.com/en/dev/ref/class-based-views/generic-display/#django.views.generic.list.ListView>

Sirviéndonos del ejemplo:

```
from django.views.generic.list import ListView  
from django.utils import timezone  
  
from articles.models import Article  
  
class ArticleListView(ListView):  
  
    model = Article  
  
    def get_context_data(self, **kwargs):  
        context = super().get_context_data(**kwargs)  
        context['now'] = timezone.now()  
        return context
```

```
from django.views.generic.list import ListView  
class PageListView(ListView):  
    model = Page
```

Ahora adaptamos la URL...

```
from .views import PageListView

urlpatterns = [
    path('', PageListView.as_view(), name='pages'),
```

Ahora vamos a recargar la web y veremos un error:

TemplateDoesNotExist at /pages/  
pages/page\_list.html

Como véis de forma automática está esperando un template llamado page\_list.html en lugar de pages.html. Tenemos dos opciones, o redefinir el atributo template\_name o cambiar el nombre del template directamente, vamos a hacer la segunda:

```
▶ pages
  ◃ page_list.html
```

Si ahora actualizamos la /pages/ veremos que no aparece ninguna página:

Playground Inicio Páginas Admin

Tranquilos, esto no significa que no lo hayamos hecho bien, sino que por defecto nuestra lista de páginas no se envía con el nombre "pages", sino con uno genérico de las ListView, y que tal como pone en el ejemplo de la documentación es:

object\_list

Por tanto sólo debemos ir a nuestro template y cambiar el nombre de esta variable:

```
{% for page in object_list %}
```

Aunque también se acepta por defecto el nombre del modelo seguido de \_list:

```
{% for page in page_list %}
```

Actualizamos de nuevo:



## Etiam pharetra risus eu venenatis sodales

Etiam pharetra risus eu venenatis sodales. Donec sit amet rhoncus odio, quis malesuada tellus. Aenean non luctus metus. Proin rutrum leo vitae tempor egestas. Phasellus mi tellus, feugiat et orci e...

[Leer más](#)

## Lorem ipsum dolor sit amet

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Sed facilisis dictum justo, a finibus mauris commodo ut. Maecenas ac ante neque. Quisque sed ultricies sapien, a accumsan elit. Sed non vive...

[Leer más](#)

Y ahí tenemos nuestra ListView. ¿Habéis visto cómo utilizando una ListView nos hemos ahorrado un montón de código? E ahí la gran utilidad de las CBV.

Vamos a hacer la DetailView para mostrar cada página de forma individual:

<https://ccbv.co.uk/projects/Django/2.0/django.views.generic.detail/DetailView>  
<https://docs.djangoproject.com/en/dev/ref/class-based-views/generic-display/#django.views.generic.detail.DetailView>

```
from django.views.generic.detail import DetailView
class PageDetailView(DetailView):
    model = Page
```

Y la URL:

```
from .views import PageListView, PageDetailView
urlpatterns = [
    path('', PageListView.as_view(), name='pages'),
    path('<int:page_id>/<slug:page_slug>', PageDetailView.as_view(), name='page'),
]
```

Evidentemente ahora tendremos que cambiar algunas cosas para que funcione. Si visitamos una página y analizamos el error veremos como solucionarlo:

AttributeError at /pages/2/etiam-pharetra-risus-eu-venenatis-sodales/  
Generic detail view PageDetailView must be called with either an object pk or a slug.

Lo que nos está indicando es que nuestra DetailView está esperando un campo `pk` o un campo `slug` para buscar la página. Nuestro objeto no tiene un campo `slug`, sólo lo estamos utilizando de adorno, así que pasaremos la clave primaria,

pk, que no es más que el id. De hecho ya lo estamos pasando, pero no con el nombre que él espera que es pk:

```
path('<int:pk>/<slug:page_slug>/',
```

Si todo va bien el siguiente paso será cambiarle el nombre al template por uno genérico:

# TemplateDoesNotExist

## pages/page\_detail.html

Así que vamos a cambiarlo:

```
<> page_detail.html
```

Si actualizamos la página veremos que esta vez ya nos funciona sin cambiar el nombre de la variable:

Playground   Inicio   Páginas   Admin

### **Etiam pharetra risus eu venenatis sodales**

Etiam pharetra risus eu venenatis sodales. Donec sit amet rhoncus odio, quis malesuada tellus. Aenean non luctus metus. Proin rutrum leo vitae tempor egestas. Phasellus mi tellus, feugiat et orci et, pellentesque elementum augue. Duis tincidunt commodo mauris sit amet posuere. Sed at suscipit est. Vivamus non mi purus. Sed vel velit in ligula aliquam sodales ac id dui. Nulla sit amet nisl velit. Proin eleifend, tellus nec fringilla vulputate, sem dolor bibendum felis, nec luctus augue risus et massa. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;

[Editar](#)

Esto es porque de forma similar a la ListView, en su template se acepta tanto la variable {{ object }} como el nombre del modelo para hacer referencia al objeto, y como tenemos {{ page }} pues ya funciona:

```
{{object.title}}
```

Y de esta forma hemos vuelto a ahorrarnos 3 líneas apenas poniendo dos palabras: model = Page

¿Supongo que ya empezáis a tener otra visión de las CBV no? Pues esto no es nada, en la siguiente lección vamos a crear nuestras propias vistas CRUD (Create-Read-Update-Delete) para gestionar páginas sin necesidad de utilizar el panel de administrador, ya veréis que útil.

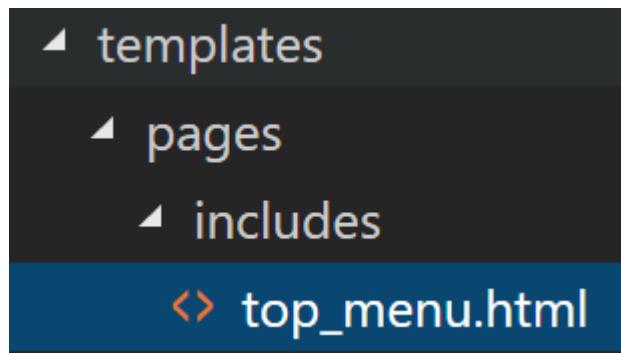
# Vistas CRUD con CBV: CreateView

Así que nuestro objetivo es crear nuestras propias vistas CRUD para manejar Páginas de forma alternativa al admin.

Lo primero que necesitamos es un lugar donde poner estas posibles acciones así que vamos a crear un submenú superior para darle la opción al usuario de crear una página en cualquier momento. Obviamente este menú sólo aparecerá si el usuario es parte del staff y está identificado.

Como la idea es contar con este submenú en cualquier parte de la app /pages/ y sería redundante añadirlo en todos los templates os voy a enseñar cómo utilizar el Tag `{% include %}` para incluir porciones de código.

Es muy fácil, vamos a crear un nuevo template `pages_menu.html` dentro de un subdirectorio `includes`. No es obligatorio hacerlo así pero es una buena forma de organizar los includes:



El contenido os lo voy a dar hecho, es básicamente un submenú con varios enlaces. Os adjunto el código en un enlace en los recursos, podéis copiarlo como yo: <https://gist.github.com/hcosta/143950b40dda8cf640076a629cec7f09>

Ahora sólo tenemos que incluir este menú en las páginas donde queramos mostrarlo, por ahora `pages_detail` y `page_list`, justo donde empieza el bloque `content`:

```
{% block content %}
{% include 'pages/includes/pages_menu.html'%}
<main role="main">
```

Si recargamos la página y estemos identificados con un usuario miembro del staff nos aparecerá este submenú dándonos la opción de crear una nueva página:

## Administrar *Crear una nueva página*

Ahora nos toca implementar la funcionalidad, para ello vamos a crear una nueva CBV de tipo CreateView:

<https://ccbv.co.uk/projects/Django/2.0/django.views.generic.edit/CreateView/>  
<https://docs.djangoproject.com/en/dev/ref/class-based-views/generic-editing/#django.views.generic.edit.CreateView>

Como siempre vamos a tomar el ejemplo de la documentación:

```
from django.views.generic.edit import CreateView
from myapp.models import Author

class AuthorCreate(CreateView):
    model = Author
    fields = ['name']
```

Y lo vamos a adaptar para manejar páginas. En nuestro caso queremos que el usuario pueda establecer el título y el contenido así que indicaremos esos campos:

```
from django.views.generic.edit import CreateView
class PageCreate(CreateView):
    model = Page
    fields = ['title', 'content', 'order']
path('create/', PageCreate.as_view(), name='page_create'),
```

Ahora vamos a cambiar la URL en el menú y a ver cuál es el siguiente paso:

```
<a class="nav-link" href="{% url 'page_create' %}"><i>Crear una nueva página</i></a>
```

Como véis en esta ocasión nos devuelve que necesita un template llamado page\_form:

**TemplateDoesNotExist at /pages/create/  
pages/page\_form.html**

Así que vamos a crearlo:

 [page\\_form.html](#)

¿Y qué contenido le ponemos? Lo básico sería lo que nos muestran la documentación. También he maquetado el contenido de antemano, así que podéis copiar el que os dejo en los recursos, igual que voy a hacer yo:

<https://gist.github.com/hcosta/456df0769f153a2458e711d3f78b75a2>

Si recargamos la página veremos un formulario sencillo, pero creado automáticamente:

**Título:**

**Contenido:**

**Orden:**

**Crear página**

Sé lo que estáis pensando. Mola, ¿pero cómo le añado mis propios estilos? Tranquilos, lo haremos más adelante. Por ahora vamos a crear una página a ver si funciona o tenemos que hacer algo más:

**Título:**

Página de prueba

Este es un contenido de prueba para un formulario creado con CBV.

**Contenido:**

Al enviar el formulario veréis que da un error:

ImproperlyConfigured at /pages/create/

No URL to redirect to. Either provide a url or define a get\_absolute\_url method on the Model.

Lo que nos está diciendo es que no hemos definido una url de redirección, pero curiosamente sí vamos manualmente a la lista de páginas veremos que se ha creado!

## Página de prueba

Este es un contenido de prueba para un formulario creado con CBV.

[Leer más](#) | [Editar](#)

Así que básicamente tenemos que volver a la View y añadir un atributo success\_url con la página que queramos redireccionar. Sabéis que me gusta hacer las cosas bien así que usaremos reverse para volver a la lista de páginas:

```
from django.urls import reverse  
success_url = reverse('pages')
```

Sin embargo esto no nos va a funcionar:

```
django.core.exceptions.ImproperlyConfigured:
```

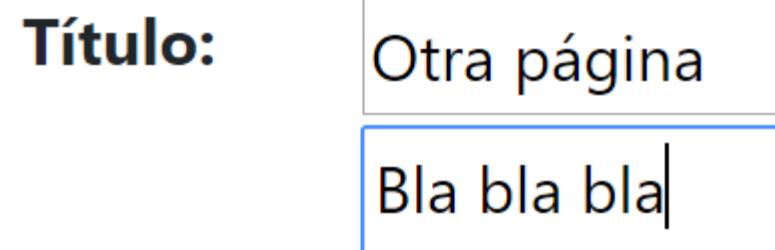
Dentro de un CBV de tipo formulario no podemos usar el reverse, la forma correcta es redefinir su método get\_success\_url y devolver la url:

```
class PageCreate(CreateView):  
    model = Page  
    fields = ['title', 'content']  
  
    def get_success_url(self):  
        return reverse('pages')
```

Que buena gente estos de Django eh, siempre poniéndonos las cosas fáciles. Por suerte si sois unos vagos y no queréis redefinir el método get\_success\_url siempre podéis hacer lo de antes sustituyendo reverse por reverse\_lazy:

```
from django.urls import reverse_lazy\n\n\nclass PageCreate(CreateView):\n    model = Page\n    fields = ['title', 'content']\n    success_url = reverse_lazy('pages')
```

Si ponemos de nuevo el servidor en marcha y creamos otra página ya debería funcionarnos:



# Otra página

Bla bla bla

[Leer más](#) | [Editar](#)

Por cierto, si queréis cambiar el orden de las páginas, podéis hacer uso de los TemplateTags dictsort y reversed para cambiarlo a vuestro gusto:

```
{% for page in page_list|dictsort:"id" reversed %}
```

Y con esto acabamos por ahora. ¿Qué os ha parecido la vista CreateView, útil verdad? Pues todavía nos faltan dos más: UpdateView y DeleteView, las veremos en la siguiente lección.

# Vistas CRUD con CBV: UpdateView

Las vistas Update y Delete nos permiten editar y borrar instancias, pero a diferencia de Create a estas tendremos que pasárselas una pk o un slug para que puedan recuperar la instancia, de forma similar a como hicimos con DetailView.

Vamos a empezar por UpdateView:

<https://ccbv.co.uk/projects/Django/2.0/django.views.generic.edit/UpdateView/>  
<https://docs.djangoproject.com/en/dev/ref/class-based-views/generic-editing/#django.views.generic.edit.UpdateView>

```
from django.views.generic.edit import CreateView, UpdateView  
\  
class PageUpdate(UpdateView):  
    model = Page  
    fields = ['title', 'content', 'order']  
\  
path('update/<int:pk>/', PageUpdate.as_view(), name='page_update'),
```

Ahora sólo tendríamos que añadir un enlace de edición en la lista de páginas para que nos lleve al formulario:

```
<p><a href="{% url 'page' page.id page.title|slugify %}">Leer más</a>  
  {% if request.user.is_staff %}  
    | <a href="{% url 'page_update' page.id %}">Editar</a>  
  {% endif %}<br>  
</p>
```

Leer más | Editar

Si probamos a editar una página, como veréis funciona directamente:

<b>Título:</b>	<input type="text" value="Otra página"/>
<b>Contenido:</b>	<input type="text" value="Bla bla bla"/>
<b>Orden:</b>	<input type="text" value="0"/>

**Crear página**

Esto es porque tanto el formulario de creación como el de edición se toman del mismo template page\_form.html. El problema es que como nuestro botón de envío tiene un texto específico “Crear página”. Hay muchísimas formas de solucionar esto, pero la genérica es simplemente utilizar otro template con el texto del botón cambiado a “Actualizar página”.

Para indicarle a nuestra UpdateView que utilice un template alternativa sólo tenemos que establecer su atributo template\_name\_suffix:

```
template_name_suffix = '_update_form'
```

Esto irá a buscar automáticamente un template page\_update\_form.html, así que vamos a clonar el que tenemos con este nombre y le cambiaremos el texto al botón:

Como en el botón de la parte inferior

page\_update\_form.html

```
{% block title %}Editar página{% endblock %}  
<input type="submit" value="Editar página" />
```

Si intentamos editar de nuevo una página ya nos saldrá bien, y si intentamos

## Editar página

actualizarla...\\

Veremos como nuevamente nos pide configurar una url de redirección:

ImproperlyConfigured at /pages/update/4/  
No URL to redirect to. Either provide a url or define a get

En este caso si quisiéramos redireccionar a la lista de páginas bastaría con hacer como antes:

```
success_url = reverse_lazy('pages')
```

Pero si queréis redirigir de nuevo al formulario es un poco más complejo. Necesitaríamos extraer la pk de la instancia. Para lograrlo esta vez sí debemos redefinir el método `get_success_url` y, ahí obtener la id de `self.object` y pasarlala como argumento a `reverse_lazy`:

```
def get_success_url(self):
    return reverse_lazy('page_update', args=[self.object.id])
```

Como esto no le da ninguna retroalimentación al usuario, os sugiero añadir un campo GET al final con un ok y capturarlo en el template:

```

def get_success_url(self):
    return reverse_lazy('page_update', args=[self.object.id]) + "?ok"
\\
<div>
    {% if 'ok' in request.GET %}
        <p style="color: green;">
            Página editada correctamente.
            <a href="{% url 'page' page.id page.title|slugify %}">Haz clic aquí para ver el resultado</a>.
        </p>
    {% endif %}
    <form action="" method="post">{% csrf_token %}

```

Página editada correctamente. Haz clic aquí para ver el resultado.

Y con esto tenemos nuestra UpdateView.

En la próxima lección veremos como trabajar con DeleteView para borrar instancias.

## Vistas CRUD con CBV: DeleteView

Bien, antes de ponernos con la DeleteView vamos a añadir un enlace para borrar páginas en nuestra lista, justo al lado de editar:

```
<p><a href="{% url 'page' page.id page.title|slugify %}">Le
  {% if request.user.is_staff %}
    | <a href="{% url 'page_update' page.id %}">Editar</a>
    | <a href="{% url 'page_delete' page.id %}">Borrar</a>
  {% endif %}
</p>
```

Ahora vamos a crear la vista:

<https://ccbv.co.uk/projects/Django/2.0/django.views.generic.edit/DeleteView/>  
<https://docs.djangoproject.com/en/dev/ref/class-based-views/generic-editing/#django.views.generic.edit.DeleteView>

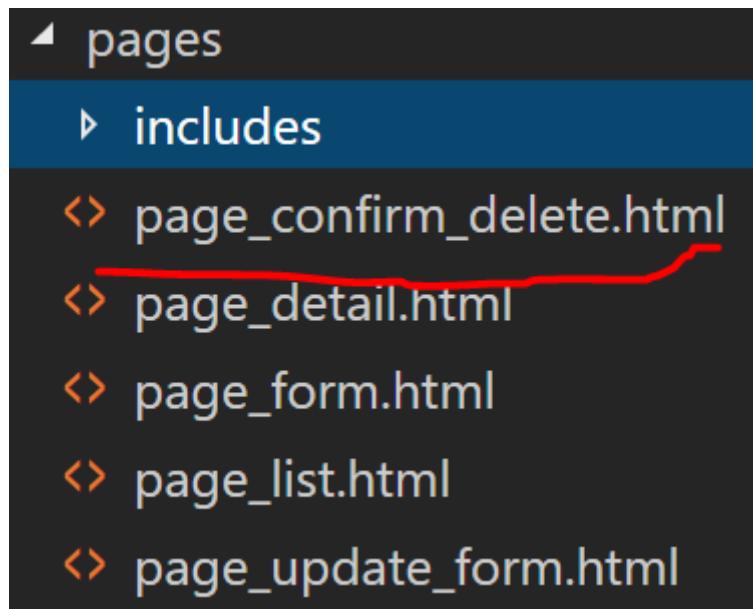
Esta es muy sencilla. Al igual que UpdateView está esperando un campo pk o slug en el Path para recuperar la instancia. Luego mostrará un formulario para confirmar el borrado y redirecciará a una success\_url.

```
from django.views.generic.edit import CreateView, UpdateView, DeleteView
\
class PageDelete(DeleteView):
    model = Page
    success_url = reverse_lazy('pages')
\
path('delete/<int:pk>', PageDelete.as_view(), name='page_delete'),
```

Ahora si probamos a borrar una página debería indicarnos el template a crear:

# TemplateDoesNotExist at pages/page\_confirm\_delete.html

Como véis este deberá llamarse page\_confirm\_delete, así que vamos a crearlo:



Como siempre os voy a facilitar el contenido ya creado tomando como referencia el de la documentación, lo podéis copiar del enlace de recursos tal y como hago

yo: <https://gist.github.com/hcosta/586f1354e662a25d43a1b2e66403eda9>

Con esto ya podríamos borrar páginas:

¿Estás seguro de que quieres borrar "Otra página"?

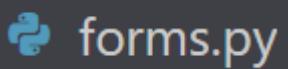
Sí, borrar la página

Como véis en no muchas lecciones hemos aprendido lo esencial sobre las CBV, hemos creado vistas CRUD para manejar modelos y tenemos todo lo necesario para gestionar páginas sin necesidad de acceder al panel de administración.

Por cierto, os comenté que os enseñaría a personalizar el formulario de las CBV, os lo enseño en la próxima lección.

# Formularios para Modelos en CBV

Personalizar el formulario enlazado a un modelo es muy fácil. Sólo tenemos que crear un formulario como hicimos con el de contacto, dentro del fichero forms.py:



Sin embargo en esta ocasión en lugar de definir cada campo a mano, vamos a hacer que Django los genere automáticamente. Para ello indicaremos el modelo dentro de una clase Meta, así como la lista campos que queremos generar y que abstraeremos de la vista:

```
from django import forms
from .models import Page

class PageForm(forms.ModelForm):
    class Meta:
        model = Page
        fields = ['title', 'content', 'order']
```

Ahora de vuelta a las vistas que manejan formularios, simplemente indicaremos el atributo form\_class y le pasaremos este formulario. Además ahora podemos borrar el campo fields porque ya viene incluido en el formulario:

```
from .forms import PageForm
\

class PageCreate(CreateView):
    model = Page
    success_url = reverse_lazy('pages')
    form_class = PageForm
```

Si accedemos a la página para crear páginas, valga la redundancia, no habrá cambiado nada y continuará funcionando sin problemas:

**Título:**

**Contenido:**

**Orden:**

**Crear página**

Ahora viene la parte interesante, pues al estar trabajando en un formulario extendido podemos añadir nuestros queridos campos Widget con configuraciones extendidas:

```
widgets = {
    'title': forms.TextInput(attrs={'class':'form-control'}),
    'order': forms.NumberInput(attrs={'class':'form-control'}),
}
\  'content':
```

```
from django import forms

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

**Título:**

**Contenido:**

**Orden:**

También podemos esconder las labels sobreescribiendo el diccionario labels y poniendo sus valores vacíos:

```
labels = {
    'title': '',
    'order': '',
    'content': '',
}
```

¿Y sabéis lo mejor de todo? Hasta podemos establecer el widget de CKeditor:

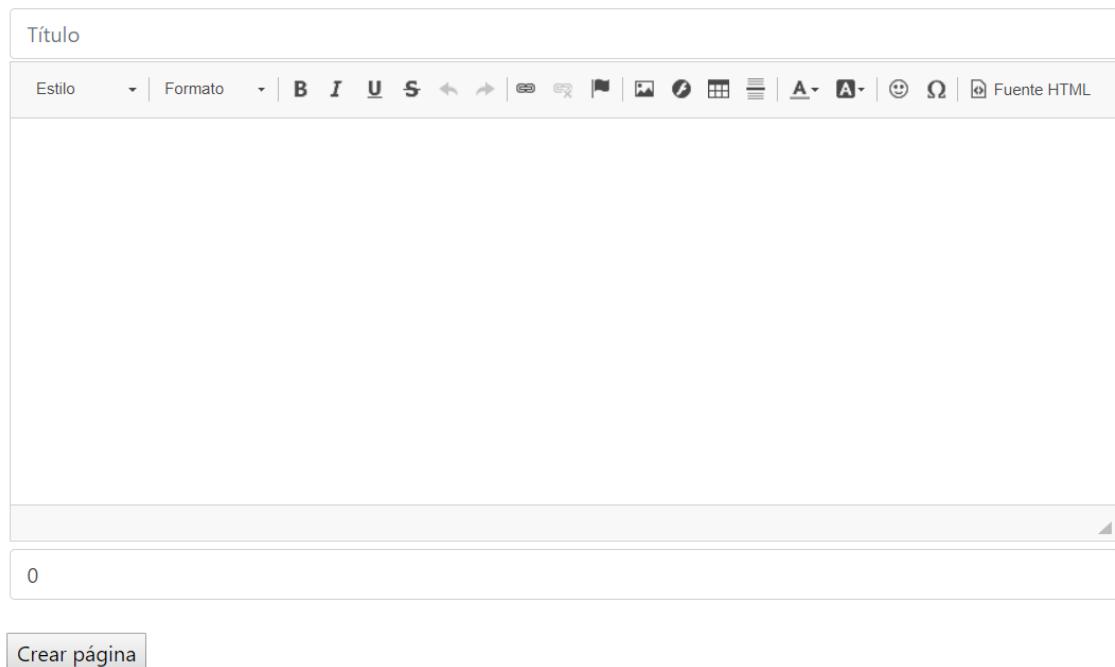
```
from ckeditor.widgets import CKEditorWidget
    'content': CKEditorWidget(),
```

Sin embargo no basta sólo con añadirlo en el form, para que aparezca debemos cargar los ficheros estáticos tal como indican en la documentación: <https://github.com/django-ckeditor/django-ckeditor#outside-of-django-admin>

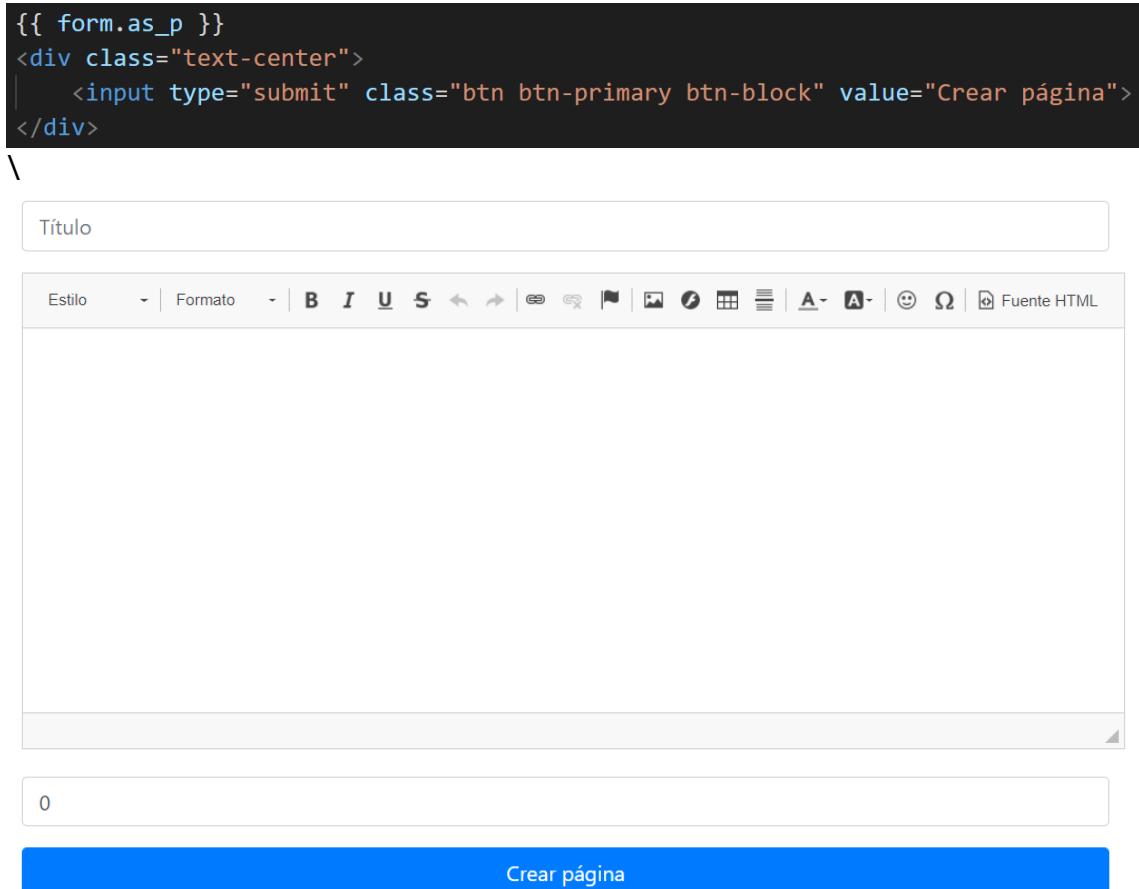
Podríamos hacerlo en el pages\_menu.html, a fin de cuentas es sólo cargar scripts y así lo tendremos centralizado:

```
{% load static %}
<script type="text/javascript" src="{% static "ckeditor/ckeditor-init.js" %}"></script>
<script type="text/javascript" src="{% static "ckeditor/ckeditor.js" %}"></script>
{% if request.user.is_staff %}
<nav class="navbar navbar-expand-lg navbar-light bg-light">
```

Con esto tenemos un formulario genial funcionando sin problemas:



Sólo nos faltaría corregir pequeños detalles, como poner la disposición en párrafos y cambiar el botón de envío por uno más elegante:



Ahora un truco de veterano. ¿Os habéis fijado que el widget de CKeditor no es adaptativo en el ancho? ¿Ni en nuestro editor de páginas, ni tampoco en el panel de administrador? Para arreglarlo necesitaremos redefinir el código CSS del widget, pero no os preocupéis porque os voy a enseñar una forma muy fácil de hacerlo. Os adjunto un Gist con el código que vamos a modificar, no es mucho.

<https://gist.github.com/hcosta/15ae0835e5824685d46e75f49efc1bcb>

Lo primero es crear el CSS que sustituirá el del widget original. Simplemente vamos a crear dentro de la app pages un directorio static/pages/css con un fichero llamado por ejemplo custom\_ckeditor.css. Dentro pondremos lo siguiente:

```
.django-ckeditor-widget, .cke_editor_id_content {  
    width: 100% !important;  
    max-width: 821px;  
}
```

A continuación tendremos que cargar este CSS siempre que mostremos el editor. Podemos cargarlo de la forma tradicional justo debajo de los scripts de ckeditor en el page\_menu.html:

```
{% load static %}  
<script type="text/javascript" src="{% static "ckeditor/ckeditor-init.js" %}"></script>  
<script type="text/javascript" src="{% static "ckeditor/ckeditor/ckeditor.js" %}"></script>  
<link href="{% static 'pages/css/custom_ckeditor.css' %}" rel="stylesheet">
```

Si reiniciamos el servidor y recargamos veremos que el textarea ya es adaptativo:

Pellentesque vitae massa ut sapien semper dictum

Estilo | Formato | **B** *I* U ~~S~~ ← → |

🔗 ↻ 🎉 | 🖼 ⚡ 📈 📋 | A A | 😊 Ω |

⊕ Fuente HTML

*Pellentesque vitae massa ut sapien semper dictum. Curabitur ut dignissim felis. Sed gravida neque nec neque feugiat, quis consequat augue aliquet. Praesent metus mi, ornare ac fringilla quis, dignissim vitae metus. Praesent a massa quis est rhoncus tristique eu imperdiet eros. Vivamus sagittis iaculis orci, ac dignissim lorem faucibus rutrum. Aliquam auctor dolor eu varius tincidunt. Sed tempor eget massa vel condimentum. Donec et rutrum orci, a imperdiet mauris.*

0

Editar página

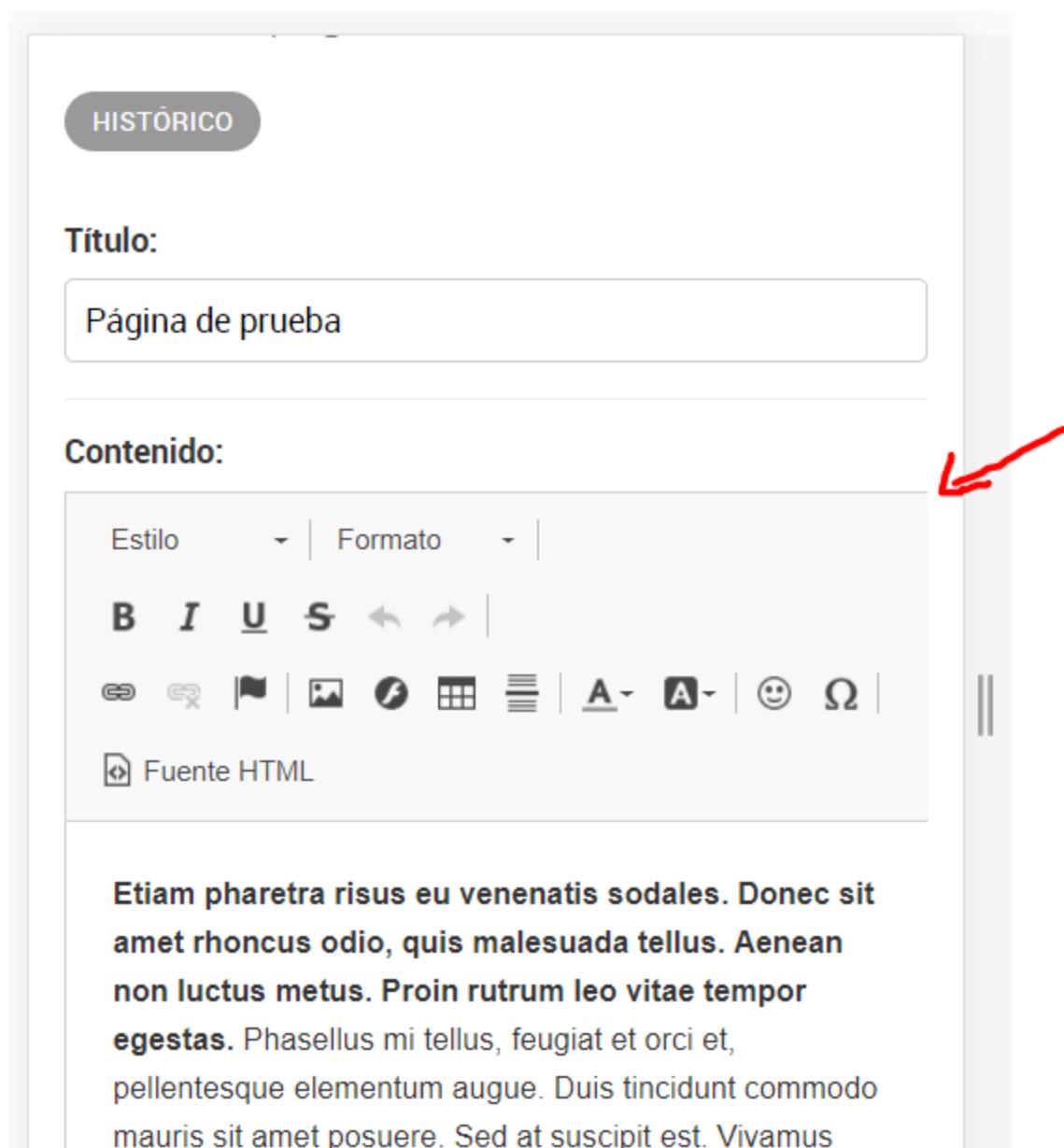
A red arrow points to the bottom-right corner of the CKEditor's content area, indicating the location where the user can click to resize the editor window.

Ahora tenemos que solucionar el del administrador. ¿Habrá alguna forma fácil o tendremos que modificar los templates? Pues tranquilos, existe una forma

extremadamente sencilla que podemos definir en el propio admin.py de la app pages:

```
class Media:  
    css = {  
        'all': ('pages/css/custom_ckeditor.css',)  
    }
```

Ahora si volvemos al admin, veréis que también es adaptativo:



Ya sólo nos falta adaptar el formulario de UpdateView para que muestre todo igual que esta:

```
class PageUpdate(UpdateView):
    model = Page
    form_class = PageForm
    template_name_suffix = '_update_form'
```

Así como el template (podemos copiar todo el form y sólo cambiar el texto del botón):

```
<div>
  {% if 'ok' in request.GET %}
    <p style="color:green;">Página editada correctamente.</p>
  {% endif %}
  <form action="" method="post">{% csrf_token %}
    {{ form.as_p }}
    <div class="text-center">
      <input type="submit" class="btn btn-primary btn-block" value="Editar página">
    </div>
  </form>
</div>
```

Y de igual forma podríamos copiar los estilos del botón para ponerlos en el formulario de borrado:

```
<input type="submit" class="btn btn-primary btn-block" value="Sí, borrar la página">
```

¿Estás seguro de que quieres borrar "Pellentesque vitae massa ut sapien semper dictum"?

Sí, borrar la página

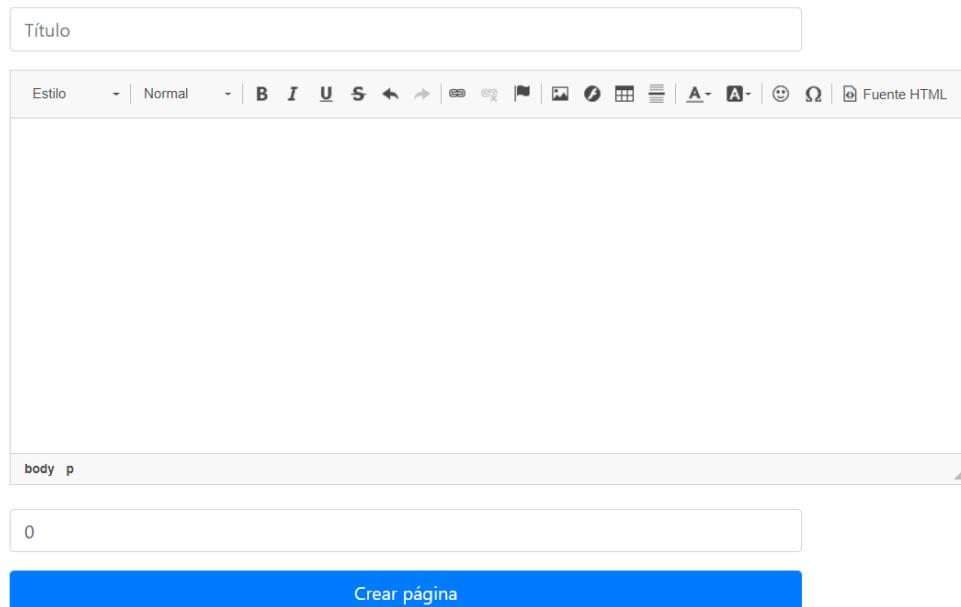
Con esto tenemos todas las vistas CRUD bien bonitas y acabamos esta interesante lección. En los recursos os dejo los enlaces a la documentación sobre Formularios y django-ckeditor por si queréis seguir aprendiendo.

<https://docs.djangoproject.com/en/dev/topics/forms/> \ <https://github.com/django-ckeditor/django-ckeditor>

## Creando un Mixin de identificación

Nuestra app de páginas está muy bien y todo funciona sin problemas, pero tenemos un fallo capital, extremadamente importante. ¿Sabéis cual es? Probad lo siguiente.

Si vamos al panel de admin y cerramos la sesión, al volver al apartado de páginas está claro que no nos aparece el menú de administrar páginas. ¿Todo bien no? Pero y si accedemos manualmente a la url /pages/create/ ?



Pues que cosas... parece que carga el formulario ¿Nos dejará crear una página?

Sí, y no sólo podemos crear, también podemos editar y borrar.

Este es nuestro fallo, cualquiera puede manejar la base de datos a través de nuestras vistas CRUD porque son PÚBLICAS.

¿Cómo vamos a solucionarlo? Pues añadiendo un acceso privado para que sólo los usuarios identificados y miembros del STAFF puedan gestionarla.

¿Recordáis el método GET que tienen las CBV? ¿Que se encargaba de la respuesta a una petición GET?. Pues resulta que hay un método que en lugar de controlar la respuesta permite controlar la petición. Ese método es dispatch.

Si redefinimos dispatch podremos comprobar la request, y dentro de la request si hay un usuario identificado:

```
def dispatch(self, request, *args, **kwargs):
    print(request.user)
    return super(PageCreate, self).dispatch(request, *args, **kwargs)
```

AnonymousUser AnonymousUser

Pero esto es sólo un debug, lo que deberíamos hacer es redireccionar al visitante al LOGIN para que se identifique:

```
def dispatch(self, request, *args, **kwargs):
    if not request.user.is_staff:
        return redirect(reverse_lazy('admin:login'))
    return super(PageCreate, self).dispatch(request, *args, **kwargs)
```

Y ahora viene la gran pregunta. ¿Debemos sobreescribir el dispatch en ~~tooooooodas~~ las vistas donde queramos que el usuario se identifique? Y la respuesta es: por supuesto que no! Para eso existen los Mixins.

Un Mixin es una implementación de una o varias funcionalidades para una clase, podemos crearlo una vez y heredar su comportamiento donde queramos dándole prioridad a su implementación antes que la de otra clase.

Fíjate, vamos a trasladar nuestro dispatch a un nuevo mixin, podemos crearlo en el propio views.py:

```
class StaffRequiredMixin(object):
    """
    Este Mixin requerirá que el usuario sea miembro del staff
    """

    def dispatch(self, request, *args, **kwargs):
        if not request.user.is_staff:
            return redirect(reverse_lazy('admin:login'))
        return super(StaffRequiredMixin, self).dispatch(
            request, *args, **kwargs)
```

Esta es nuestra funcionalidad especial, ahora sólo debemos hacer que nuestras CBV hereden de él dándole prioridad durante la herencia, es decir, poniendo más a la izquierda:

```
class PageCreate(StaffRequiredMixin, CreateView): ...
class PageUpdate(StaffRequiredMixin, UpdateView): ...
class PageDelete(StaffRequiredMixin, DeleteView): ...
```

Esto básicamente sobreescribirá el método dispatch del CreateView con el dispatch de nuestro Mixin, y así tendremos nuestra funcionalidad implementada. Vamos a probarlo...

## Administración de Django

Nombre de usuario:

hector

Contraseña:

.....

Iniciar sesión

Con esto hemos aprendido como se crea y utiliza un Mixin, si queréis más información os dejaré documentación al respecto, tened en cuenta que podéis sobreescribir cualquier atributo o método, por lo que son muy útiles:\ <https://docs.djangoproject.com/en/dev/topics/class-based-views/mixins/>

## Usando decoradores de identificación

¿Entonces cada vez que queramos comprobar si un usuario está identificado tenemos que crear un Mixin? Pues no, eso os lo he enseñado con la excusa de que aprendáis a crear Mixins y ver el potencial que tienen, pero por suerte para nosotros Django incluye varias funciones decoradoras para requerir identificación antes de devolver una vista.

Por ejemplo, dentro de nuestro StaffRequiredMixin podríamos ahorrarnos la parte de comprobar al usuario haciendo uso de la función decoradora staff\_member\_required a través del decorador de métodos method\_decorator:

```

from django.utils.decorators import method_decorator
from django.contrib.admin.views.decorators import staff_member_required
\

class StaffRequiredMixin(object):
    """
    Este Mixin requerirá que el usuario sea miembro del staff
    """

    → @method_decorator(staff_member_required)
    def dispatch(self, request, *args, **kwargs):
        return super(StaffRequiredMixin, self).dispatch(
            request, *args, **kwargs)

```

Lo bueno que tiene este decorador es que redirecciona de nuevo a la página donde teníamos que identificarnos para seguir trabajando:

› 127.0.0.1:8000/admin/login/?next=/pages/create/

Pero en el fondo esto de la redirección es lo de menos, lo bueno que tiene el uso de decoradores es que no estamos obligados a utilizarlos en Mixins, sino que se pueden enviar a las CBV directamente, sólo debemos definir el `method_decorator` antes de la CBV e indicarle el decorador y el nombre del método donde queremos añadirlo, en nuestro caso el `dispatch` (podemos borrar el mixin porque ya no lo necesitamos):

```

@method_decorator(staff_member_required, name='dispatch')
class PageCreate(CreateView): ...

@method_decorator(staff_member_required, name='dispatch')
class PageUpdate(UpdateView): ...

@method_decorator(staff_member_required, name='dispatch')
class PageDelete(DeleteView): ...

```

Cómo véis ni siquiera necesitábamos un Mixin, pero en el proceso hemos aprendido un montón de cosas.

Existen dos funciones decoradoras más:

- `login_required` para comprobar que un usuario esté identificado
- `permission_required` para comprobar que un usuario esté identificado y a la vez tenga algún permiso

Os voy a dejar documentación de todas en los recursos, así como la forma de decorar directamente en el URLPatterns:\

\ <https://docs.djangoproject.com/en/dev/topics/auth/default/#the-login->

[required-decorator](https://docs.djangoproject.com/en/dev/topics/auth/default/#the-permission-required-decorator) \ [decorator](https://docs.djangoproject.com/en/dev/topics/class-based-views/intro/#decorating-class-based-views) \ [views/decorators/](https://docs.djangoproject.com/en/dev/_modules/django/contrib/admin/views/decorators/)

Y con esto acabamos la introducción a las CBV y las vistas CRUD.

Ahora ya contáis con los fundamentos necesarios para empezar a trabajar la autenticación, el registro y el manejo de usuarios, así que si no os perdáis las próximas lecciones.

## Refactorizando CBV en las URLs - NO SE EXPLICARÁ, DEMASIADOS PROBLEMAS

Estoy seguro de que con todo lo que hemos hecho hasta ahora sobre CBV ya os ha quedado claro su potencial, ¿pero os imagináis que pudiéramos ahorrarnos incluso más trabajo? Pues váis a flipar, porque ¿y si os digo que podemos trasladar casi todo lo que tenemos sólo a nuestras urls y ahorrarnos las vistas os lo creeríais?

Mirad, vamos a core/urls y haremos la magia ahí:

```
from django.urls import path
from django.views.generic.base import TemplateView

urlpatterns = [
    path('', TemplateView.as_view(template_name="core/home.html"), {'title': 'Mi super Web Playground'}, name='home'),
    path('sample/', TemplateView.as_view(template_name="core/sample.html"), name="sample"),
]
```

# Mi super Web Playground

CBV, autenticación, registros, perfiles y más

¿Habéis visto? Podemos generar nuestras CBV ahí mismo e incluso enviar un diccionario de contexto. ¿Pero podremos hacer lo mismo con las CBV de la app Pages? Vamos a probar:

```

from django.urls import path, reverse_lazy, reverse
from django.views.generic.list import ListView
from django.views.generic.detail import DetailView
from django.views.generic.edit import CreateView, UpdateView, DeleteView
from .models import Page
from .forms import PageForm

urlpatterns = [
    path('', ListView.as_view(
        model=Page), name='pages'),

    path('<int:pk>/<slug:page_slug>', DetailView.as_view(
        model=Page), name='page'),

    path('create/', CreateView.as_view(
        model=Page, form_class=PageForm, success_url=reverse_lazy('pages')), name='page_create'),

    path('update/<int:pk>', UpdateView.as_view(
        model=Page, form_class=PageForm, template_name_suffix='_update_form',
        success_url=reverse_lazy('pages')), name='page_update'),

    path('delete/<int:pk>', DeleteView.as_view(
        model=Page, success_url=reverse_lazy('pages')), name='page_delete'),
]

```

Para las tres primeras no tendremos ningún problema, podemos generarlas sin problemas. Lo malo es la UpdateView porque contiene un método. Los métodos no podemos redefinirlos aquí, por lo que lo más sensato es dejarnos de historias y dejarla como la teníamos:

```

from .views import PageUpdate
\path('update/<int:pk>', PageUpdate.as_view(), name='page_update'),

```

Finalmente la DeleteView si podemos generarla sin problemas:

```

path('delete/<int:pk>', DeleteView.as_view(
    model=Page, success_url=reverse_lazy('pages')), name='page_delete'),
]

```

Así que podemos volver a views.py y borrar todo lo que no necesitamos, aunque a vosotros mejor os recomiendo comentarlo por si en un futuro queréis analizarlo:

```
from django.views.generic.edit import UpdateView, DeleteView
from django.urls import reverse_lazy
from .models import Page
from .forms import PageForm

# Create your views here.
class PageUpdate(UpdateView):
    model = Page
    form_class = PageForm
    template_name_suffix = '_update_form'

    def get_success_url(self):
        return reverse_lazy('page_update', args=[self.object.id]) + "?ok"
```

Hemos pasado de tener más de 30 líneas a apenas unas 13 definiendo nuestras CBV en el propio urls.py. ¿Qué os parece?

---

## Octava App [Registration] Iniciando la sesión

Hasta ahora para iniciar y cerrar la sesión lo hemos hecho siempre a través del panel de administrador, pero esto sólo nos vale para webs sencillas. Si queremos crear algo más elaborado nos veremos obligados a integrar este proceso dentro del diseño de la página.

Por suerte gracias a las CBV no tenemos que desarrollar el sistema desde cero.

Para reutilizar parte del sistema de autenticación de Django, empezaremos creando nueva app llamada registration (se le suele dar este nombre). Esta app manejará tanto la autenticación como el registro y vamos a estar bastante lecciones con ella.

Así que como siempre, vamos a empezar creando esta nueva app:

```
python manage.py startapp registration
```

En esta app no vamos a programar casi nada, sólo se va a encargar de almacenar algunos templates.

A continuación vamos a dirigirnos al fichero urls.py del proyecto base y vamos a configurar un nuevo apartado /accounts/, debe ser ese nombre pues es el que las CBV de Django implementan por defecto:

```
# Paths de auth
path('accounts/', include('django.contrib.auth.urls')),
```

Sólo con esto se nos han dado de alta varias URLs para manejar la autenticación. Podemos ver exactamente cuales accediendo al servidor

127.0.0.1:8000/accounts/  
/accounts/:\\

```
accounts/ login/ [name='login']
accounts/ logout/ [name='logout']
accounts/ password_change/ [name='password_change']
accounts/ password_change/done/ [name='password_change_done']
accounts/ password_reset/ [name='password_reset']
accounts/ password_reset/done/ [name='password_reset_done']
accounts/ reset/<uidb64>/<token>/ [name='password_reset_confirm']
accounts/ reset/done/ [name='password_reset_complete']
```

Básicamente tenemos el login, el logout y varias vistas para manejar la gestión de contraseñas. Por ahora centrémonos en las dos primeras. Son todas las AUTH VIEWS que podemos ver en nuestra web favorita:

[https://ccbv.co.uk/\\](https://ccbv.co.uk/)

## AUTH VIEWS

[LoginView](#)

[LogoutView](#)

[PasswordChangeDoneView](#)

[PasswordChangeView](#)

[PasswordResetCompleteView](#)

[PasswordResetConfirmView](#)

[PasswordResetDoneView](#)

[PasswordResetView](#)

Podríamos haber creado todas una a una manualmente, pero al haber hecho el include directo nos lo hemos ahorrado.

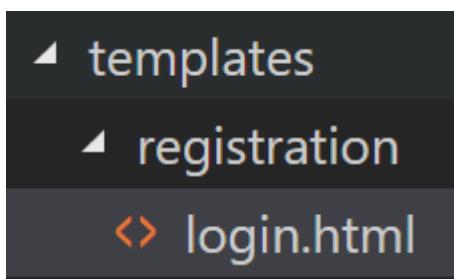
Sea como sea si intentamos acceder a /accounts/login/ cargaremos una LoginView genérica:

**127.0.0.1:8000/accounts/login/**

Y obviamente nos dará error de template:

**TemplateDoesNotExist at /accounts/login/  
registration/login.html**

Tenemos que crear el template login.html en el directorio templates/registration de la app que hemos creado antes:



Dentro sólo debemos preparar y mostrar un formulario de la forma clásica, pero como quiero ahorraros trabajo he preparado un diseño de antemano. Simplemente he copiado el código HTML generado por defecto, he comprobado errores globales en la lista form.non\_field\_errors y he añadido los atributos class y placeholder a los inputs.

<https://gist.github.com/hcosta/7c3a46d5e5d061d936031ad5a6654acd>

Una vez creado el template tenemos que activar la app registration para que se cargue y reiniciar el servidor:

**'registration',**

Y si actualizamos la página deberías ser capaces de verla correctamente:

## Iniciar sesión

Si intentamos acceder, veréis que nos da un fallo:

## Page not found (404)

**Request Method:** GET

**Request URL:** http://127.0.0.1:8000/accounts/profile/

Esto ocurre porque Django está redireccionandonos automáticamente a una supuesta página de perfil en /accounts/profile. Más adelante crearemos un perfil de usuario, así que por ahora nos interesa cambiar de alguna forma esta redirección. ¿Cómo lo hacemos? Pues yendo a settings.py y añadiendo:

```
# Auth redirects  
LOGIN_REDIRECT_URL = 'home'
```

Ahora si nos identificamos correctamente podréis notar como nos redirecciona a la vista portada que tiene ese nombre:

# Mi super Web Playground

CBV, autenticación, registros, perfiles y más

Si en lugar de home quisiéramos redirigir al usuario a la lista de páginas podríamos hacer con:

```
LOGIN_REDIRECT_URL = 'pages:pages'
```

Ya que esta opción actúa como la función reverse.

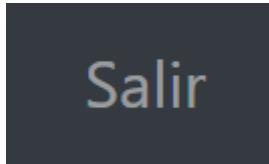
Vamos a tomarnos una pausa y luego hacemos el logout en la próxima lección.

# Cerrando la sesión

En este punto sería interesante modificar el diseño del menú superior para mostrar estas opciones en lugar de las del administrador. Vamos a volver a base.html:

```
<ul class="navbar-nav">
    {% if not request.user.is_authenticated %}
        <li class="nav-item">
            <a class="nav-link" href="{% url 'login' %}">Acceder</a>
        </li>
    {% else %}
        <li class="nav-item">
            <a class="nav-link" href="{% url 'logout' %}">Salir</a>
        </li>
    {% endif %}
</ul>
```

Al estar identificados debería aparecernos la opción de Salir:



Salir

Si salimos nos llevará a la pantalla por defecto como si saliéramos desde el administrador.

## Sesión terminada

Gracias por el tiempo que ha dedicado hoy al sitio web.

[Iniciar sesión de nuevo](#)

Obviamente no queremos esto y como podéis suponer podemos cambiar la página de redirección de la misma forma que hicimos con el Login:

```
# Auth redirects
LOGIN_REDIRECT_URL = 'pages:pages'
LOGOUT_REDIRECT_URL = 'home'
```

Con esto acabamos la autenticación ¿Ha sido bastante fácil no? En las siguientes lecciones estaremos trabajando el registro y la recuperación de contraseñas.

## Registro con CBV (1)

La parte del registro no es tan simple como la autenticación, para ello sí que debemos crear una vista y manejar nosotros la lógica, pero Django nos ayudará bastante, ya veréis.

Antes de nada vamos a incluir las URLs de registration en el urls.py global y así nos olvidamos de ello:

```
# Paths de auth
path('accounts/', include('django.contrib.auth.urls')),
path('accounts/', include('registration.urls')),
```

Fijaros que las añadiremos justo debajo de las que hicimos para la autenticación. Esto no es ningún problema, ya que simplemente se extenderán las que teníamos antes.

Bien, ahora vamos a crear nuestra vista SignUp para manejar el registro. Fijaros muy bien lo que vamos a hacer:

```
from django.contrib.auth.forms import UserCreationForm
from django.views.generic import CreateView
from django.urls import reverse_lazy

class SignUpView(CreateView):
    form_class = UserCreationForm
    success_url = reverse_lazy('login')
    template_name = 'registration/signup.html'
```

En lugar de crear un formulario desde cero lo que haremos es importar uno genérico (UserCreationForm) y se lo pasaremos a una vista CreateView para que lo maneje todo automáticamente, por eso os decía que Django nos ayudaría bastante.

Vamos a crear el formulario. En este caso no he preparado un diseño de antemano porque quiero enseñaros cómo modificarlo dinámicamente, así que vamos a tomar de referencia el de registro y modificarlo:

```

{% extends 'core/base.html' %}
{% load static %}
{% block title %}Registro{% endblock %}
{% block content %}
<main role="main">
    <div class="container">
        <div class="row mt-3">
            <div class="col-md-9 mx-auto mb-5">
                <form action="" method="post">{% csrf_token %}
                    <h3 class="mb-4">Registro</h3>
                    {{form.as_p}}
                <p>
                    <input type="submit" class="btn btn-primary btn-block" value="Confirmar">
                </p>
            </form>
        </div>
    </div>
</main>
{% endblock %}

```

Finalmente configuraremos la URL, vamos a crear un fichero urls.py:

```

from django.urls import path
from .views import SignUpView

urlpatterns = [
    path('signup/', SignUpView.as_view(), name='signup'),
]

```

Ahora sólo tenemos que acceder a accounts/signup y ver la magia:

---

**127.0.0.1:8000/accounts/signup/**

---

## Registro

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

Contraseña:

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

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

**Confirmar**

¡Increíble! ¿Funcionará?

## Registro

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

Contraseña:

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

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

**Confirmar**

Pues parece que sí, porque nos ha redireccionado a login.

Os aconsejo redireccionar a otra página o hacer la típica modificación pasando un parámetro GET, por lo menos así añadimos retroalimentación al usuario. Por desgracia no podemos concatenar en el success\_url así que deberemos sobreescribir el método get\_success\_url y devolver nuestra cadena desde ahí:

```
class SignUpView(CreateView):
    form_class = UserCreationForm
    template_name = 'registration/signup.html'

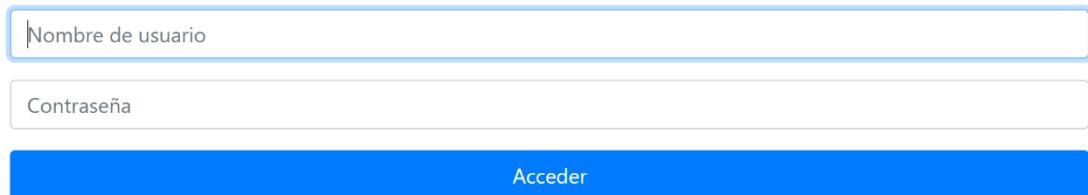
    def get_success_url(self):
        return reverse_lazy('login') + '?register'

<div class="col-md-9 mx-auto mb-5">
    {%
        if 'register' in request.GET %}
        <p style="color: green;">
            Usuario registrado correctamente, ya puedes identificarte.
        </p>
    {%
        endif %}
    <form action="" method="post">{%
        csrf_token %}
```

Probamos de nuevo:

Usuario registrado correctamente, ya puedes identificarte.

## Iniciar sesión



The screenshot shows a simple login form with two input fields and a button. The first field is labeled "Nombre de usuario" and the second is labeled "Contraseña". Below the fields is a blue button labeled "Acceder".

Bien, vamos a mejorar la apariencia del formulario. Podríamos editar el HTML pero perderíamos un montón de validaciones automáticas, así que vamos a aprovecharnos de esta vista que tenemos para modificar los widgets en tiempo de ejecución. Para hacerlo debemos saber qué nombre tienen los campos, así vamos a observar el formulario generado y los atributos name de los inputs:

```
<p><label for="id_username">Nombre de usuario:</label> <input type="text" name="username" maxlength="150" autofocus required id="id_username" /> <span class="helptext">Requerido. 150 caracteres como máximo. Únicamente letras, dígitos y @/./+/-/_ </span></p>
<p><label for="id_password1">Contraseña:</label> <input type="password" name="password1" required id="id_password1" /> <span class="helptext"><ul>
<li>Su contraseña no puede asemejarse tanto a su otra información personal.</li><li>Su contraseña debe contener al menos 8 caracteres.</li>
<li>Su contraseña no puede ser una clave utilizada comúnmente.</li><li>Su contraseña no puede ser completamente numérica.</li></ul></span></p>
<p><label for="id_password2">Contraseña (confirmación):</label> <input type="password" name="password2" required id="id_password2" /> <span class="helptext">Para verificar, introduzca la misma contraseña anterior.</span></p>
<p>
```

En nuestro caso el formulario tiene tres: username, password1 y password2.

Sabiendo esto vamos a la vista y haremos lo siguiente para recuperar el formulario:

```
def get_form(self, form_class=None):
    form = super(SignUpView, self).get_form()
    return form
```

El método get\_form obtiene el formulario antes de devolverlo, de manera que podemos modificar sus widgets. Fíjate:

```
from django import forms
```

```

def get_form(self, form_class=None):
    form = super(SignUpView, self).get_form()
    form.fields['username'].widget = forms.TextInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Nombre de usuario'})
    form.fields['password1'].widget = forms.PasswordInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Contraseña'})
    form.fields['password2'].widget = forms.PasswordInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Repita la contraseña'})
    form.fields['username'].label = ''
    form.fields['password1'].label = ''
    form.fields['password2'].label = ''
    return form
\

def get_form(self, form_class=None):
    form = super(SignUpView, self).get_form()
    form.fields['username'].widget = forms.TextInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Nombre de usuario'})
    form.fields['password1'].widget = forms.PasswordInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Contraseña'})
    form.fields['password2'].widget = forms.PasswordInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Repita la contraseña'})
    form.fields['username'].label = ''
    form.fields['password1'].label = ''
    form.fields['password2'].label = ''
    return form

```

## Registro

Nombre de usuario:

Requerido. 150 caracteres como máximo. Únicamente letras, dígitos y @/.+/-/\_

Contraseña:

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

Contraseña (confirmación):

Para verificar, introduzca la misma contraseña anterior.

[Confirmar](#)

Ya sólo tenemos que esconder las labels, podríamos hacerlo de forma similar a como hemos cambiado los widgets... pero es mucho más rápido ir al template y añadir una tag <style> para esconderlas todas de golpe:

```
<style>label{display:none}</style>
```

## Registro

Nombre de usuario

Requerido. 150 caracteres como máximo. Únicamente letras, dígitos y @./+/-/\_

Contraseña

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

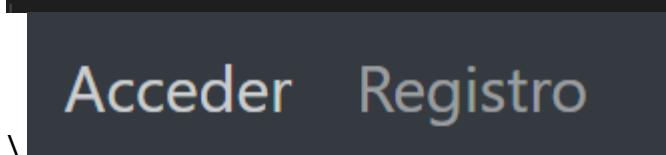
Ripete la contraseña

Para verificar, introduzca la misma contraseña anterior.

Confirmar

Finalmente para dejarlo perfecto podríamos añadir el enlace al registro en el menú superior:

```
{% if not request.user.is_authenticated %}  
    <li class="nav-item">  
        <a class="nav-link" href="{% url 'login' %}">Acceder</a>  
    </li>  
    <li class="nav-item">  
        <a class="nav-link" href="{% url 'signup' %}">Registro</a>  
    </li>  
{% else %}
```



Con esto tenemos un registro básico para seguir trabajando en las próximas lecciones.

## PEmail obligatorio al registrarse

Nuestro registro está muy bien, pero hay un problema, y es que por defecto Django no requiere el email para dar de alta una cuenta. Esto podría parecer una nimiedad, pero sin un email no os puedo enseñar cómo configurar la restauración de contraseña, y es un tema importante porque le añade un plus de calidad a nuestras páginas.

Podríamos crear el formulario desde cero o extender UserCreationForm. Mi pereza me impide trabajar más de la cuenta así que haremos lo segundo.

```

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

class UserCreationFormWithEmail(UserCreationForm):
    email = forms.EmailField(required=True,
        help_text="Requerido. 255 caracteres como máximo y debe ser un email válido.")

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

```

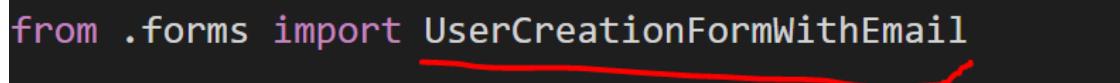
Tened en cuenta que el campo email no es adrede, sino que es el nombre del campo en el modelo User aunque no se haga uso de él por defecto.

Ahora tenemos que adaptar la vista:

```

from .forms import UserCreationFormWithEmail

```



```

class SignUpView(CreateView):
    form_class = UserCreationFormWithEmail
    template_name = 'registration/signup.html'

    def get_success_url(self):
        return reverse_lazy('login') + '?register'

```

Y sólo nos faltará extender el método get\_form con el nuevo campo email:

```

def get_form(self, form_class=None):
    form = super(SignUpView, self).get_form()
    form.fields['username'].widget = forms.TextInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Nombre de usuario'})
    form.fields['email'].widget = forms.EmailInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Dirección email'})
    form.fields['password1'].widget = forms.PasswordInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Contraseña'})
    form.fields['password2'].widget = forms.PasswordInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Repita la contraseña'})
    form.fields['username'].label = ''
    form.fields['email'].label = ''
    form.fields['password1'].label = ''
    form.fields['password2'].label = ''
    return form

```

```

def get_form(self, form_class=None):
    form = super(SignUpView, self).get_form()
    form.fields['username'].widget = forms.TextInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Nombre de usuario'})
    form.fields['email'].widget = forms.EmailInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Dirección email'})
    form.fields['password1'].widget = forms.PasswordInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Contraseña'})
    form.fields['password2'].widget = forms.PasswordInput(attrs={'class': 'form-control mb-2', 'placeholder': 'Repita la contraseña'})
    form.fields['username'].label = ''
    form.fields['email'].label = ''
    form.fields['password1'].label = ''
    form.fields['password2'].label = ''
    return form

```

Un apunte importante: Tened en cuenta que estamos extendiendo el formulario UserCreationForm y éste contiene sus propias validaciones. La única forma de sustituir los widgets sin afectar al resto del formulario es modificarlos en tiempo de ejecución desde el método get\_form.

En todo caso nuestro formulario se ve bastante bien... ¿Funcionará? Vamos a probarlo:

## Registro

manolo

Requerido. 150 caracteres como máximo. Únicamente letras, dígitos y @/.+/-/\_

manolo@hola.com

Requerido. 150 caracteres como máximo y debe ser un email válido.

.....

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

.....

Para verificar, introduzca la misma contraseña anterior.

Confirmar

Parece que sí, vamos a verificar su email a través del panel de administrador:

**Usuario registrado correctamente, ya puedes identificarte.**

Y sí, se ha guardado el correo:

manolo

manolo@hola.com

Sin embargo todavía tenemos un último problema por solucionar... Lo veremos en la próxima lección.

## Validando un email único

Pues sí, el problema es que pueden haber dos usuarios con el mismo email. Carece de toda lógica, pero el registro por defecto de Django no sólo considera que el email es un campo optativo, sino que tampoco es único. Menudo despropósito...

Probad a crear otro usuario con el mismo email, ya veréis:

<input type="checkbox"/>	manolo	manolo@hola.com
<input type="checkbox"/>	sdsdsdsd	manolo@hola.com

Hay innumerables formas de solucionar este problema, cada una más compleja que las demás, por eso vamos a utilizar la más sencilla de todas: añadir una validación específica de campo. Sólo tenemos que ir al formulario y añadir un método para validar, o mejor dicho “limpiar” el campo email:

```
def clean_email(self):
    email = self.cleaned_data.get('email')
    if User.objects.filter(email=email).exists():
        raise forms.ValidationError(u'El email ya está registrado, prueba con otro.')
    return email
```

La lógica es sobreescribir el método `clean_<nombre_del_campo>`, de manera que a través de `self` podemos acceder a los campos después de que Django los valide y añadir una validación extra. Si no hay ningún Usuario con ese email devolveremos el email validado, pero si existe alguno invocaremos un error de tipo `ValidationError` con el mensaje que queremos mostrar en el formulario:

## Registro

manolete

Requerido. 150 caracteres como máximo. Únicamente letras, dígitos y @./+/-/\_

- El email ya está registrado, prueba con otro.

manolo@hola.com

Requerido. 150 caracteres como máximo y debe ser un email válido.

Contraseña

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

Repita la contraseña

Para verificar, introduzca la misma contraseña anterior.

Confirmar

Y con esto acabamos el formulario de registro.

## Restaurar contraseña

Los usuarios son despistados por naturaleza, se registran en muchos sitios y luego no recuerdan sus credenciales, así que es necesario proveerles de una forma para recuperar su cuenta en caso de que olviden su contraseña. Sin embargo para que puedan hacerlo es necesario que cuenten con su correo electrónico enlazado al usuario, y esa es la razón por la que nos hemos pasado las últimas dos lecciones configurando el email en el registro.

Así que lo primero será configurar una cuenta de correo en Django como hicimos durante la web empresarial para enviar correos de prueba, podríamos volver a utilizar vuestra bandeja de Mailtrap, pero en esta ocasión os voy a mostrar una alternativa por si algún día no contáis con Internet, se trata de configurar un servidor SMTP para pruebas, éste hará ver que envía los correos pero en realidad los guardará en forma de ficheros dentro de un directorio, vamos a configurarlo:

```
# Emails
if DEBUG:
    EMAIL_BACKEND = "django.core.mail.backends.filebased.EmailBackend"
    EMAIL_FILE_PATH = os.path.join(BASE_DIR, "sent_emails")
else:
    # Aquí hay que configurar un email real para producción
    pass
```

De esta forma nuestros emails se crearán en un nuevo directorio llamado `sent_emails`, obviamente podéis poner cualquier otro nombre. Por cierto, si en lugar de crear ficheros queréis mostrar los correos por la terminal también podéis, sólo tenéis que cambiar `filebased` por `console`. Os adjunto documentación por si queréis aprender otras opciones de configuración: <https://docs.djangoproject.com/en/dev/topics/email/#console-backend> <https://docs.djangoproject.com/en/dev/topics/email/#smtp-backend>

Ahora antes de ponernos manos a la obra, dejadme deciros que este proceso de restauración ya nos funciona, lo está gestionando el panel de administrador automáticamente. Probad a acceder la URL de restauración `/accounts/password_reset/`:

Restablecer contraseña

¿Ha olvidado su clave? Introduzca su dirección de correo a continuación y le enviaremos por correo electrónico las instrucciones para establecer una nueva.

Correo electrónico:

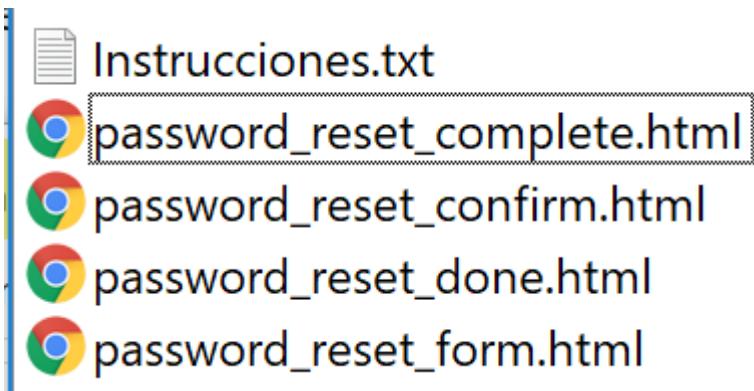
[Restablecer mi contraseña](#)

Si ya os está bien que aparezca el admin podríais dejarlo así, pero creo que es más interesante utilizar nuestros propios templates. Así que tendremos que empezar diciéndole a Django que dé prioridad a los templates de la app `Registration` antes que los del panel de admin. Simplemente debemos poner nuestra app por delante en `settings.py`:

```
'registration',
'django.contrib.admin',
```

Si no hacéis este cambio nunca os aparecerán vuestros propios templates.

Bien, vamos a por esta primera página password\_reset. Lo único que debemos hacer es crear nuestra propia versión del template bajo el nombre password\_reset\_form.html. Como tendremos que crear varios os los voy a adjuntar todos en un descargable descargar este y todos los demás en los recursos.



Una vez tengáis los nuevos templates en el directorio /templates/registration/ si accedemos de nuevo....

## Restaurar contraseña

Para poder restaurar la contraseña debes indicar el email asociado a tu cuenta de usuario.

Introduce tu email

Confirmar

Ahora si introducimos un correo, esté o no enlazado a un usuario, se redireccionará a una página de confirmación en la url [password\\_reset/done/](#), probad con uno de mentira:

Para poder restaurar la contraseña debes indicar el email asociado a tu cuenta de usuario.

dsssd@sdssd.com

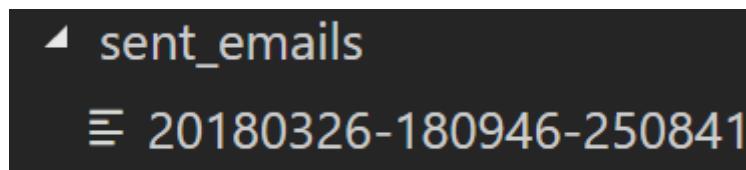
Confirmar

## Restablecimiento de contraseña enviado

Le hemos enviado por email las instrucciones para restablecer la contraseña, si es que existe una cuenta con la dirección electrónica que indicó. Debería recibirlas en breve.

Si no recibe un correo, por favor asegúrese de que ha introducido la dirección de correo con la que se registró y verifique su carpeta de spam.

Ahora repetid el proceso pero utilizando un email real que hayáis utilizado para registraros anteriormente, se os debería crear el directorio `sent_mails` en la raíz del proyecto:



Lo que tendremos dentro es un fichero con el email que “en teoría” se ha enviado al usuario. Si lo abrimos podremos seguir las instrucciones:

<http://127.0.0.1:8000/accounts/reset/NQ/4ut-73fb6025ffee936e2ddc/>

Su nombre de usuario, en caso de haberlo olvidado: manolo

Como véis se envía un enlace con un token para recuperar la contraseña, así como un recordatorio del nombre del usuario. Vamos a abrir ese enlace a ver qué nos aparece:

### Escriba la nueva contraseña

Por favor, introduzca su contraseña nueva dos veces para verificar que la ha escrito correctamente.

Introduce la nueva contraseña

Repite la nueva contraseña

Cambiar mi contraseña

Como véis aquí tenemos el formulario de restauración que se toma del template `password_reset_confirm.html`, en él debemos introducir la contraseña nueva dos veces. Probad a poner una contraseña numérica o equivocaros apostando, os deberían aparecer los errores de validación, y si ponéis una correcta nos llevará a la última pantalla que nos falta por ver, `password_reset_complete`:

# Restablecimiento de contraseña completado

Su contraseña ha sido establecida. Ahora puede seguir adelante e iniciar sesión.

[Iniciar sesión](#)

Si tenéis curiosidad sobre cómo son los templates originales os voy a dejar un enlace al repositorio oficial: <https://github.com/django/django/tree/master/django/contrib/admin/templates/registration>

Por cierto, existen un par de vistas más para cambiar la contraseña del usuario autenticado, pero esas las veremos cuando creemos los perfiles.

Lo único que nos falta es añadir el enlace de restauración en el formulario de autenticación:

```
</form>
<p>¿Ha olvidado su clave? Puede restaurarla <a href="{% url 'password_reset' %}">aquí</a>.</p>
```

## Iniciar sesión

The screenshot shows a login form with two input fields: 'Nombre de usuario' (Username) and 'Contraseña' (Password), both with placeholder text. Below the fields is a large blue button labeled 'Acceder' (Log in). Above the 'Acceder' button, there is a link '¿Ha olvidado su clave? Puede restaurarla aquí.' (Forgot your password? You can restore it here.)

¿Ha olvidado su clave? Puede restaurarla [aquí](#).

Con esto acabamos la parte de la autenticación y el registro.

## Creando un perfil de usuario

Nuestro siguiente objetivo es ofrecerle al usuario tres campos extras para personalizar su perfil: una imagen para usarla de avatar, un texto como biografía y un enlace a una página web.

Podríamos crear un nuevo modelo User con estos tres campos extra, pero esa sería la solución compleja. Vamos a por lo simple, crear un modelo Profile que contenga esos 3 campos y una relación 1:1 con un Usuario, de manera que cada instancia de User tenga enlazada una de Profile.

Lo vamos a crear en nuestra app Registration ¿Porqué ahí en lugar de en una nueva app? Bueno, no sé si lo recordaréis pero por defecto django al registrarse el usuario lo redireccionaba a /accounts/profile/ ¿no? Pues esa es la razón. Más adelante sí crearemos una app Profiles para los perfiles públicos, pero esta parte privada la manejaremos en registration.

En este modelo no necesitamos darle nombre a los campos porque no vamos a usarlos desde el panel de administrador:

```
from django.db import models
from django.contrib.auth.models import User

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    avatar = models.ImageField(upload_to='profiles', null=True, blank=True)
    bio = models.TextField(null=True, blank=True)
    link = models.URLField(max_length=200, null=True, blank=True)
```

Lo más remarcable es el uso de la relación OneToOneField. Esta es la tercera relación que podemos crear en Django y le indica al modelo que sólo puede haber perfil para cada usuario, no se pueden tener dos perfiles para un mismo usuario, ni distintos usuarios para un mismo perfil, esa es la clave del OneToOne. Haciendo un breve repaso, en Django tenemos tres tipos de relaciones:

- OneToOneField, relación (1:1) de 1 a 1. Única para ambos sentidos: 1 usuario - 1 perfil
- ForeignKeyField, relación (1:N) de 1 a varios. Única en un sólo sentido: 1 autor - N entradas
- ManyToManyField, relación (N:M) de varios a varios. No única en ningún sentido: N entradas - M categorías

Dicho esto, recordad que tenemos un campo de imagen, así que debemos tener instalado Pillow en el entorno virtual y configurar el servidor para servir ficheros MEDIA. En mi caso ya tengo Pillow, así que sólo tengo que configurar el servidor:

```
# Media Files
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

```
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)
```

Perfecto.

Ahora necesitamos crear una página para editar el perfil. Por ahora vamos a crearla como una simple TemplateView, luego os explico porqué. También

utilizaremos el decorador login\_required en el dispatch, ya que no se puede editar un perfil sin estar identificado:

```
from django.views.generic.base import TemplateView
from django.utils.decorators import method_decorator
from django.contrib.auth.decorators import login_required

@method_decorator(login_required, name='dispatch')
class ProfileUpdate(TemplateView):
    template_name = "registration/profile_form.html"
```

El template vamos a copiarlo de page\_form y lo adaptaremos un poco:

```
{% extends 'core/base.html' %}
{% load static %}
{% block title %}Perfil{% endblock %}
{% block content %}
<main role="main">
    <div class="container">
        <div class="row mt-3">
            <div class="col-md-9 mx-auto mb-5">
                <h3>Perfil</h3>
                <form action="" method="post">{% csrf_token %}
                    {{ form.as_p }}
                    <div class="text-center">
                        <input type="submit" class="btn btn-primary btn-block" value="Actualizar">
                    </div>
                </form>
            </div>
        </div>
    </div>
</main>
{% endblock %}
```

Y configuraremos el enlace en urls.py:

```
from .views import SignUpView, ProfileUpdate

urlpatterns = [
    path('signup/', SignUpView.as_view(), name='signup'),
    path('profile/', ProfileUpdate.as_view(), name='profile'),
]
```

Finalmente pondremos el enlace en nuestro menú cuando el usuario inicia la sesión:

```
{% else %} -----  
  <li class="nav-item">  
    | <a class="nav-link" href="{% url 'profile' %}">Perfil</a>  
  </li> -----  
    | <li class="nav-item">  
    |   | <a class="nav-link" href="{% url 'logout' %}">Salir</a>  
    | </li>  
  {% endif %}
```

Y comentamos la redirección de settings para el login:

```
# LOGIN_REDIRECT_URL = 'pages:pages'
```

Una vez hecho todo esto vamos a identificarnos con un usuario a ver si nos lleva al perfil:

## Perfil

Actualizar

Bien, con esto bastará, vamos a tomarnos un descanso y continuamos en la siguiente lección.

## Haciendo editable el perfil

La razón por la que no hemos creado directamente una UpdateView la vais a ver en breve, por ahora vamos a migrar la app registration:

```
python manage.py makemigrations registration  
\  
python manage.py migrate registration
```

A continuación vamos a transformar la TemplateView en una UpdateView, importando el modelo Profile:

```
from django.views.generic.base import TemplateView  
from django.views.generic.edit import UpdateView  
\ from .models import Profile
```

```
@method_decorator(login_required, name='dispatch')
class ProfileUpdate(UpdateView):
    model = Profile
    fields = ['avatar', 'bio', 'link']
    success_url = reverse_lazy('profile')
```

Ahora la gran pregunta... ¿cómo conseguimos la primary key del perfil para editarla y enviarla a la CBV UpdateView? Pues os voy a contestar con otra pregunta, ¿sería buena idea pasar una pk del perfil en el path? Obviamente la respuesta es NO, si hiciéramos eso cualquiera podría editar todos los perfiles sólo sabiendo el id. Además, ¿si todavía no hemos creado una instancia de perfil como vamos a mostrar un formulario para editarla?

Por suerte aunque no sepamos el id del perfil, podemos saber el id del Usuario autenticado sin necesidad de pasarlo en el path, pues éste se almacena en la propia request.

Ahora la última pieza que nos falta es recuperar el perfil a partir del id del usuario que hay en request. ¿Cómo haremos esta magia oscura? Pues sobreescritiendo el método get\_object de nuestra UpdateView, encargada de recuperar el objeto que se tiene que modificar:

```
@method_decorator(login_required, name='dispatch')
class ProfileUpdate(UpdateView):
    model = Profile

    def get_object(self):
        return Profile.objects.get(user=self.request.user)
```

Get\_object debe devolver el objeto que editaremos, así que simplemente podemos recuperarlo a partir de self.request.user:

DoesNotExist at /accounts/profile/  
Profile matching query does not exist.

Sin embargo con esto no es suficiente. Y es que estamos intentando recuperar un Profile que todavía no existe. Por suerte podemos cambiar la consulta para hacer uso del método get\_or\_create en lugar de get. Lo bueno que tiene es que creará la instancia si no existe... y devuelve dos valores: la instancia y un booleano indicando si la acaba de crear:

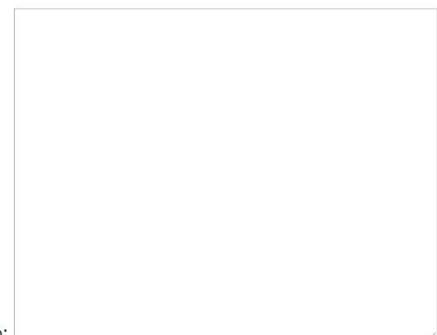
```
def get_object(self):
    profile, created = Profile.objects.get_or_create(user=self.request.user)
    return profile
```

Tened en cuenta que aunque sólo utilicemos el primer valor, debemos declarar los dos porque se están devolviendo ambos y hay que guardarlos en algún lado.

Con esto vamos a probar si funciona..

## Perfil

Avatar:  Ningún archivo seleccionado



Bio:

Link:

Parece que sí, incluso podemos modificar la información...

## Perfil

Avatar:  avatarhektor\_...O - copia.png

Me llamo Héctor

Bio:

Link:  https://www.hektorprofe.r

Lo único que no funciona bien es la imagen. ¿Sabéis por qué? Pues no es que tengamos un error ni mucho menos, es sólo que no hemos configurado el formulario para aceptar ficheros desde el propio HTML. Hacerlo es muy fácil, vamos a nuestro template y...

```
<form action="" method="post" enctype="multipart/form-data">{% csrf_token %}
```

Recarguemos el formulario e intentemos enviar de nuevo la imagen:

## Perfil

Me llamo Héctor

Avatar: Actualmente: [profiles/hektor\\_bQ85MN2.jpg](#)

Modificar:  Ningún archivo seleccionado

Bio:

Link:

¡Ahí la tenemos!

Por cierto, ¿sabéis que tan fácil sería acceder al perfil de cualquier usuario? Pues sólo tendríamos que hacer referencia a su profile como si fuera otro campo de User:

```
<h3>Perfil</h3>
Tu enlace: {{request.user.profile.link}}
```

Tu enlace: <https://www.hektorprofe.net/>

Esto es gracias a que las relaciones OneToOneField se enlazan automáticamente al modelo con su mismo nombre, ni siquiera tenemos que definir un related\_name, ¿no es genial?

En la siguiente lección le daremos un mejor aspecto a este formulario.

## Mejorando el formulario de perfil

En esta pequeña lección vamos a mejorar algo el aspecto del formulario de perfil, mezclando nuestro propio formulario con widgets y un poco de bootstrap.

En primer lugar vamos a implementar un formulario para nuestro modelo Profile:

```
class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['avatar', 'bio', 'link']
        widgets = {
            'avatar': forms.ClearableFileInput(attrs={'class': 'form-control-file mt-3'}),
            'bio': forms.Textarea(attrs={'class': 'form-control mt-3', 'rows': 3, 'placeholder': 'Biografía'}),
            'link': forms.URLInput(attrs={'class': 'form-control mt-3', 'placeholder': 'Enlace'}),
        }
```

Fijaros que hemos puesto un ClearableFileInput, eso es para conservar la opción de borrar la imagen:

```
from .forms import UserCreationFormWithEmail, ProfileForm
@method_decorator(login_required, name='dispatch')
class ProfileUpdate(UpdateView):
    form_class = ProfileForm
    success_url = reverse_lazy('profile')

    def get_object(self):
        try:
            return Profile.objects.get(user=self.request.user)
        except Profile.DoesNotExist:
            return Profile.objects.create(user=self.request.user)
```

Sólo con esto nuestro formulario ya tendrá un mejor aspecto:

## Perfil

Avatar: Actualmente: [profiles/19237182.jpg](#)  Limpiar

Modificar:

Seleccionar archivo Ningún archivo seleccionado

Bio:

Me llamo Héctor y me gusta programar!

Link:

<https://www.hektorprofe.net/>

Actualizar

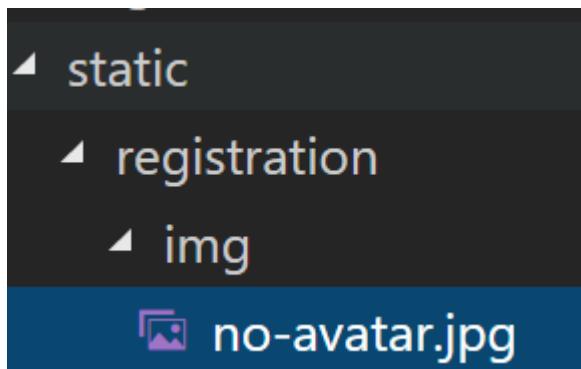
Pero todavía es muy soso, y esas opciones para la imagen... quedan muy mal.

La única salida que nos queda es maquetar cada campo manualmente, así que he preparado un diseño de antemano, os adjunto el enlace en los recursos:

<https://gist.github.com/hcosta/3302f2cf81b71872c503b405adfb1708>

Una vez pongamos el nuevo template todo tendrá mejor aspecto:

¿Bastante bien no? Pues todavía podemos mejorar una última cosa, y es que ahora mismo si nuestro usuario no tiene una imagen de perfil no nos aparece nada. ¿Y si ponemos una imagen de avatar genérica? Os adjunto en los recursos una imagen llamada no-avatar.jpg, copiadla dentro de los ficheros estáticos de la app static/registration/img:



Ahora vamos a modificar ligeramente el código para mostrar esta imagen si el usuario no tiene un avatar:

```

{% else %}
| 
{% endif %}

```

Por último recargamos el servidor para cargar en memoria los nuevos ficheros estáticos y...

Muy bien! En la próxima lección añadiremos un enlace al formulario para que el usuario pueda cambiar su email.

## Opción de editar el email

Editar el email podría ser una odisea utilizando vistas basadas en clases. Eso es porque una UpdateView está limitada a un modelo, que en nuestro caso es Profile. Si quisiéramos editar dos modelos, Profile y User , no podríamos. En su lugar deberíamos implementar una vista tradicional y manejar manualmente un formulario compartido de forma parecida a como hicimos el de contacto.

Para estos casos mi recomendación es dejarnos llevar por el framework, así que vamos a seguir la lógica de las UpdateView y vamos a separar la edición del email en un formulario a parte. Sí, puede que el usuario tengo que hacer un clic extra, pero para nosotros será mucho más fácil, y además podemos dar a entender que el email no es algo que se pueda ir cambiando tan a la ligera.

Así que se me ha ocurrido que podemos mostrar un campo con el email pero sólo de lectura y debajo un enlace que diga: Si deseas editar tu email haz clic aquí. Vamos a hacerlo.

```
 {{ form.link }}  
<input type="email" value="{{request.user.email}}" class="form-control mt-3" readonly>  
<p class="mt-3">Si deseas editar tu email haz clic <a href="{% url 'profile_email' %}">aquí</a>. </p>  
<input type="submit" class="btn btn-primary btn-block mt-3" value="Actualizar">
```

Este campo es sólo de adorno, fijaros que no tiene un atributo name, por lo que no se enviará:

The screenshot shows a Django admin-style profile edit form. It includes fields for profile picture, bio, website, and email. A note at the bottom indicates that the email field is for reading only and has a link to edit it.

Perfil

Seleccionar archivo Ningún archivo seleccionado

¿Borrar?

Me llamo Héctor y me gusta programar!

https://www.hektorprofe.net/

[REDACTED]@gmail.com

Si deseas editar tu email haz clic [aquí](#).

Actualizar

Ahora vamos a diseñar un formulario para editar el email:

```

class EmailForm(forms.ModelForm):
    email = forms.EmailField(required=True, max_length=254,
                           help_text="Requerido. 254 caracteres como máximo y debe ser un email válido.")

    class Meta:
        model = User
        fields = ['email']
        widgets = {
            'email': forms.EmailInput(attrs={'class': 'form-control-file mt-3'}),
        }

```

Pero con esto no basta, necesitamos validar el email, y en esta ocasión debemos comprobar dos cosas: primero si se ha editado el email y si es así, comprobar que éste no se repita.

¿Cómo lo hacemos? Pues por suerte para nosotros hay una lista llamada `changed_data` a la que podemos acceder desde el método `clean`, desde ahí es muy fácil comprobar si ha cambiado el email:

```

def clean_email(self):
    email = self.cleaned_data.get('email')
    if 'email' in self.changed_data:
        if User.objects.filter(email=email).exists():
            raise forms.ValidationError(u'El email ya está registrado, prueba con otro.')
    return email

```

Ahora sólo tenemos que preparar la vista:

```

from .forms import UserCreationFormWithEmail, ProfileForm, EmailForm
\

@method_decorator(login_required, name='dispatch')
class EmailUpdate(UpdateView):
    form_class = EmailForm
    success_url = reverse_lazy('profile')

    def get_object(self):
        return self.request.user

```

Vamos a crear una url:

```

from .views import SignUpView, ProfileUpdate, EmailUpdate

urlpatterns = [
    path('signup/', SignUpView.as_view(), name='signup'),
    path('profile/', ProfileUpdate.as_view(), name='profile'),
    path('profile/email/', EmailUpdate.as_view(), name='profile_email'),
]

```

Si accedemos a la URL pese a no haber creado el template, nos dará un curioso error:

TemplateDoesNotExist at /accounts/profile/email/  
auth/user\_form.html

Al contrario de lo podíamos esperar, no se va a buscar el template en la app registration, sino en auth. Eso es porque Django maneja el modelo User en esa app, pero no es nada que no podamos cambiar manualmente con el template\_name:

```
template_name = 'registration/profile_email_form.html'
```

Así que vamos a crearlo, podemos tomar de referencia el de registro (signup.html) y cambiarlo ligeramente::

```
{% extends 'core/base.html' %}  
{% load static %}  
{% block title %}Email{% endblock %}  
{% block content %}  
<style>.errorlist{color: red;} label{display:none}</style>  
<main role="main">  
    <div class="container">  
        <div class="row mt-3">  
            <div class="col-md-9 mx-auto mb-5">  
                <form action="" method="post">{% csrf_token %}  
                    <h3 class="mb-4">Email</h3>  
                    {{form.as_p}}  
                    <p><input type="submit" class="btn btn-primary btn-block" value="Actualizar"></p>  
                </form>  
            </div>  
        </div>  
    </div>  
</main>  
{% endblock %}
```

Vamos a ver si funciona:

## Email

hcostaguzman@gmail.cor Requerido. 254 caracteres como máximo y debe ser un email válido.

Editar

Sí, ahí lo tenemos. ¿Pero porqué no se añade correctamente el widget que hemos establecido?

Bueno, en este caso volvemos a estar ante un modelo extendido, y es que User ya tiene sus propios campos y validaciones. Para sobreescribir sus widgets

debemos hacerlo en tiempo de ejecución, así que vamos a llevarnos el widget a la vista:

```
class Meta:  
    model = User  
    fields = ['email']  
    widgets = {  
        'email': forms.EmailInput(attrs={'class': 'form-control mt-3'})  
    }  
  
\  
def get_form(self, form_class=None):  
    form = super>EmailUpdate, self).get_form()  
    form.fields['email'].widget = forms.EmailInput( attrs={'class': 'form-control mb-2', 'placeholder': 'Email'})  
    return form
```

Una vez hecho esto, ya debería aparecernos bien:

## Email

Requerido. 254 caracteres como máximo y debe ser un email válido.

[Editar](#)

Y si intentamos poner un email ya existente nos mostrará el fallo:

## Email

- El email ya está registrado, prueba con otro.

Requerido. 254 caracteres como máximo y debe ser un email válido.

[Editar](#)

Finalmente sólo falta modificar el enlace y ya lo tendremos:

```
{% url 'profile_email' %}
```

En la siguiente lección haremos algo parecido permitiendo al usuario editar su contraseña. Para lograrlo nos serviremos de las vistas automatizadas de la app auth, de forma muy similar a como hicimos el formulario para reestablecerla.

## Opción de editar la contraseña

Bien, como os comenté al final de la anterior lección vamos a dar al usuario la posibilidad de editar su contraseña. Por suerte no tenemos que hacerlo todo nosotros, la mayor parte del trabajo está hecho.

Vamos a empezar añadiendo un enlace en nuestro formulario de perfil:

```
accounts/ password_change/ [name='password_change']  
\  
<p class="mt-3">Si deseas editar tu email haz clic <a href="{% url 'profile_email' %}>aquí</a>.   
<br>Y si quieres cambiar tu contraseña haz clic <a href="{% url 'password_change' %}>aquí</a>.   
</p>
```

Si deseas editar tu email haz clic [aquí](#).

Y si quieres cambiar tu contraseña haz clic [aquí](#).

Si accedemos al enlace nos aparecerá el formulario del administrador:

Cambio de contraseña

Por favor, introduzca su contraseña antigua, por seguridad, y después introduzca la nueva contraseña dos veces para verificar que la ha escrito correctamente.

Contraseña antigua:

Contraseña nueva:

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

Contraseña nueva (confirmación):

[CAMBIAR MI CONTRASEÑA](#)

Sólo tenemos que cambiar estos templates por los nuestros. Os los voy adjuntar en los recursos, como siempre sólo tenéis que copiarlos en el directorio templates de la app registration:

## Nombre

 Instrucciones.txt

 password\_change\_done.html

 password\_change\_form.html

Vamos a probar:

## Cambio de contraseña

Por favor, introduzca su contraseña antigua por seguridad, y después introduzca dos veces la nueva contraseña para verificar que la ha escrito correctamente.

Cambiar mi contraseña

## Contraseña cambiada correctamente

Puedes volver a tu perfil haciendo clic [aquí](#).

Para crearlos he tomado como referencia los originales, os vuelvo a dejar el enlace al repositorio por si queréis ojearlos: <https://github.com/django/django/tree/master/django/contrib/admin/templates/registration>

Con esto nuestro formulario está oficialmente acabado, hay un par de detalles que vamos a mejorar en las siguientes dos lecciones y que utilizaré como excusa para introducirlos un nuevo concepto como son las señales y también las pruebas unitarias.

## Introducción a las señales

El primer detalle que vamos a solucionar es la prevención de un potencial error que podría hacer tambalear toda nuestra página, y ese es el momento de crear un perfil.

Tal y como lo tenemos ahora, siempre que un usuario se registre y se identifique será redireccionado directamente a su perfil, de manera que se creará la instancia enlazada. Pero, ¿qué ocurriría si por alguna razón el usuario se registra y no accede nunca? Pues que nos quedará un usuario sin perfil, una situación tendiente a errores en el futuro.

Así que vamos a solucionar esta situación introduciendo algo muy interesante conocido como Signals o Señales.

Una señal es un disparador que se llama automáticamente después de un evento que ocurre en nuestro ORM. En nuestro caso lo que haremos es crear automáticamente un perfil justo después de que se cree un usuario, y así

estaremos 100% seguros de que todos los usuarios cuentan con un perfil desde el primer momento.

Crear una señal es relativamente fácil, lo haremos en el fichero registration/models.py:

Dentro vamos a crear una función que enlazaremos a través de una señal al modelo User justo después de crear un usuario, tiene que recibir un sender, una instance y los kwargs. Instance hace referencia al objeto que envía la señal, y que en nuestro caso será el usuario recién creado, así que simplemente creamos el perfil y nos quedamos tan anchos:

```
def ensure_profile_exists(sender, instance, **kwargs):
    Profile.objects.get_or_create(user=instance)
    print("Se acaba de crear un usuario y su perfil enlazado")
```

Ahora debemos debemos transformar la función en una señal. Lo haremos con el decorador @receiver, al cual le pasaremos un tipo de señal, en nuestro caso post\_save (después del guardado), y un sender, que corresponderá al modelo encargado de enviar la señal, en nuestro caso User:

```
from django.dispatch import receiver
from django.db.models.signals import post_save
\

@receiver(post_save, sender=User)
def ensure_profile_exists(sender, instance, **kwargs):
    Profile.objects.get_or_create(user=instance)
    print("Se acaba de crear un usuario y su perfil enlazado")
```

Con esto ya lo tenemos, pero hay que tener en cuenta que esta señal se ejecutará siempre que se guarde la instancia, ¿habrá alguna forma de ejecutar la consulta sólo cuando se crea por primera vez? Pues sí. Si comprobamos del diccionario kwargs la clave created, esta debería devolver True si justo después de crearse la instancia se ha llamada a la señal. Si no existe devolveremos False por defecto:

```
@receiver(post_save, sender=User)
def ensure_profile_exists(sender, instance, **kwargs):
    if kwargs.get('created', False):
        Profile.objects.get_or_create(user=instance)
        print("Se acaba de crear un usuario y su perfil enlazado")
```

Vamos a registrar un nuevo usuario a ver si funciona ha creado su perfil...

# Registro

pepito

Requerido. 150 caracteres

pepito@pepito.com

Se acaba de crear un usuario y su perfil enlazado

Pues parece que sí, pero para estar seguros vamos a meternos un momento en la shell y a consultarla manualmente. La shell es un entorno python y django dentro de nuestro proyecto, de manera que podemos interactuar en tiempo real a través de instrucciones:

```
(django2) C:\Users\hcost\Documents\CursoDjangoFinalizado\webplayground>python manage.py shell
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from registration.models import Profile
>>> Profile.objects.get(user__username='pepito4')
<Profile: Profile object (2)>
>>>
```

Como véis el objeto existe, por lo que podemos afirmar que efectivamente se ha creado el perfil.

Sin embargo utilizar la shell para hacer estas comprobaciones muy tedioso, en la siguiente lección vamos a ver cómo realizar una prueba unitaria que cree un usuario y después compruebe que se ha creado el perfil.

Por cierto, también existen las señales pre\_save, pre\_delete y post\_delete, pero como nosotros necesitamos que el usuario ya exista, hemos utilizado post\_save. Para más información podéis consultar la documentación sobre

señales, os la dejo en los recursos: <https://docs.djangoproject.com/en/dev/topics/signals/>

## Introducción a las pruebas unitarias

Vamos a crear una prueba muy simple, ésta creará un usuario de pruebas y luego comprobará que existe su perfil, si existe pasará el test y si no fallará.

Para crear un test iremos al fichero tests.py de la app registration. Ahí veremos que por defecto hay una clase cargada llamada TestCase, vamos a cargar nuestros modelos User y Profile.

```
from django.contrib.auth.models import User
from .models import Profile
```

Ahora vamos a crear la clase con las pruebas:

```
# Create your tests here.
class ProfileTestCase(TestCase):
    def setUp(self):
        pass

    def test_profile_exists(self):
        pass
```

El método setUp de TestCase es donde haremos la preparación, y luego tenemos un método propio, test\_profile\_exists, donde definiremos el test. Puede tener cualquier nombre siempre que empiece por test\_.

La preparación es muy simple, sólo debemos crear un usuario de pruebas:

```
def setUp(self):
    User.objects.create_user('test', 'test@test.com', 'test1234')
```

Normalmente los objetos se crean con su método create, pero el caso de User es especial, ya que contiene un método create\_user encargado de cifrar la contraseña por nosotros.

Ahora vamos a definir el test. Se supone que en este punto el usuario creado existirá, así que podemos comprobar que haya un perfil con ese usuario:

```
def test_profile_exists(self):
    exists = Profile.objects.filter(user__username='test').exists()
    self.assertEqual(exists, True)
```

La prueba en sí misma es muy sencilla, solo hacemos un assertEquals para comprobar que exists sea True.

Y ahora ¿cómo la ejecutamos? Pues desde el manage.py haremos:

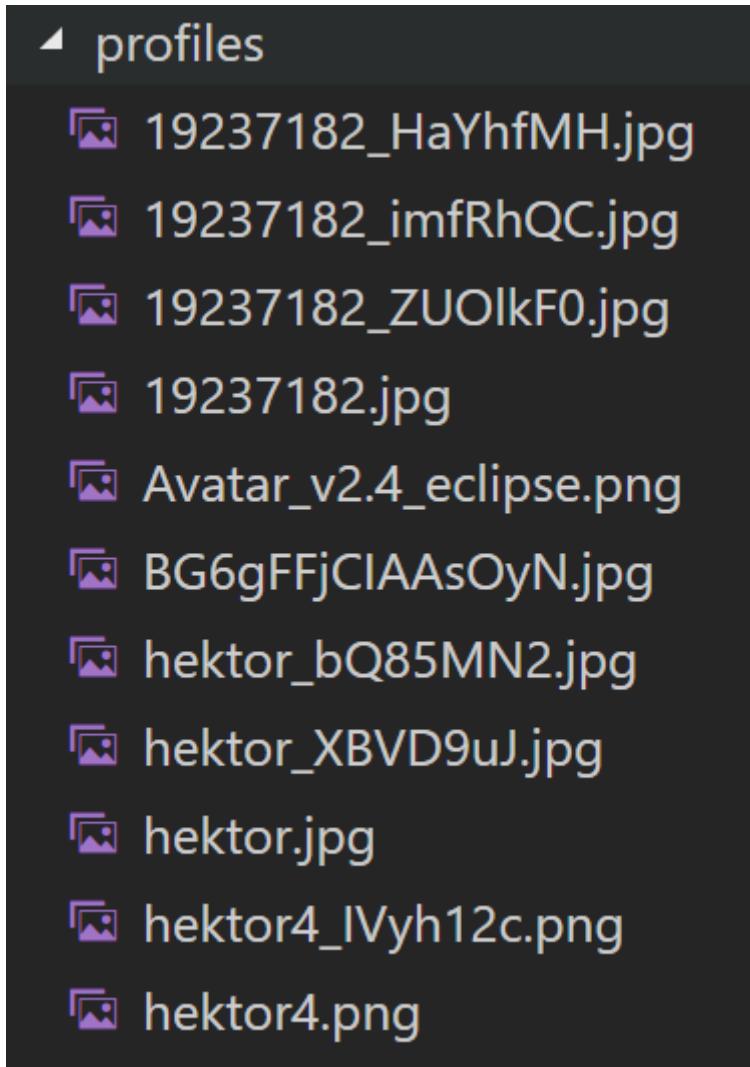
```
(django2) C:\Users\hcost\Documents\CursoDjangoFinalizado\webplayground>python manage.py test registration
Creating test database for alias 'default'...
System check identified no issues (0 silenced).
Se acaba de crear un usuario y su perfil enlazado
.
-----
Ran 1 test in 0.065s
OK
Destroying test database for alias 'default'...
```

Como véis se ha creado una base de datos de prueba, casualmente también llamada test, se ha ejecutado 1 test y ha devuelto OK. Luego se ha borrado la base de datos de prueba, así que no queda ni rastro del usuario test.

Con esto lo tenemos perfecto y podemos estar seguros de que nunca quedará un usuario sin perfil.

## Optimizando el almacenamiento del avatar

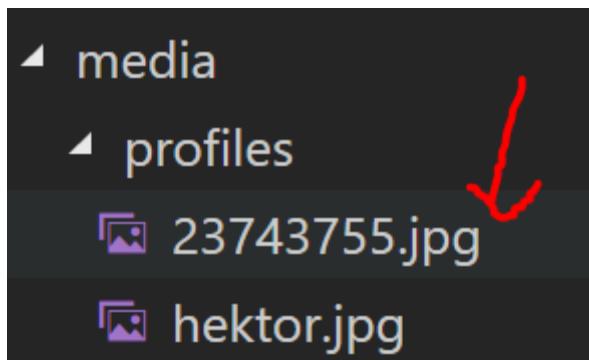
El último detalle que os sugiero optimizar es la forma de almacenar el avatar de usuario. No sé vosotros, pero yo he estado haciendo un montón de pruebas y se me han guardado un montón de imágenes diferentes:



Obviamente no nos interesa almacenar todos los avatares de cada usuario, sería un despilfarro de espacio, y el espacio vale dinero. Vamos a almacenar sólo la imagen más reciente. Para lograrlo vamos a crear nuestro propio update\_to en forma de función. Dentro recuperaremos el antiguo avatar y lo borraremos haciendo uso de su útil método delete, que comparten ambos ImageField y FileField. Justo después devolveremos la ruta al fichero de la forma normal:

```
def custom_upload_to(instance, filename):
    old_instance = Profile.objects.get(pk=instance.pk)
    old_instance.avatar.delete()
    return 'profiles/' + filename # Devolvemos la ruta nueva
```

De esta forma nos aseguraremos de almacenar una y solo una imagen por usuario, optimizando mucho la memoria. Probadlo, borrad todas las imágenes del directorio profiles y fijaros como siempre que añadimos una se borra la anterior:



Con esto damos por finalizado el formulario de perfil. La siguiente lección es una práctica en la que deberás crear una nueva app para mostrar los perfiles de forma pública.

Si queréis más información sobre la creación de funciones para `upload_to` os dejo el enlace oficial en los recursos: [https://docs.djangoproject.com/en/dev/ref/models/fields/#django.db.models.FileField.upload\\_to](https://docs.djangoproject.com/en/dev/ref/models/fields/#django.db.models.FileField.upload_to)

## Práctica: Novena App [Profiles] Perfiles públicos

Muy bien pues vamos a solucionar la práctica anterior. Como se trata de una app relativamente sencilla la he preparado de antemano y solo vamos a repasar los puntos importantes:



Vamos a empezar repasando las vistas:

```

from django.shortcuts import get_object_or_404
from django.views.generic.list import ListView
from django.views.generic.detail import DetailView
from registration.models import Profile

# Create your views here.
class ProfileListView(ListView):
    model = Profile
    template_name = 'profiles/profile_list.html'

class ProfileDetailView(DetailView):
    model = Profile
    template_name = 'profiles/profile_detail.html'

    def get_object(self):
        return get_object_or_404(Profile, user__username=self.kwargs['username'])

```

No tienen mucha complicación, sólo remarcar que por defecto los templates se van a buscar a la app registration, donde está creado el modelo Profile, simplemente debemos cambiar el template\_name.

```

from django.urls import path
from .views import ProfileListView, ProfileDetailView

profiles_patterns = [
    path('', ProfileListView.as_view(), name='list'),
    path('<username>/', ProfileDetailView.as_view(), name='detail'),
], "profiles")

```

Respecto a las URLs en este caso yo he utilizado la lógica de app:vista, no era obligatorio hacerlo así, por lo que mientras os funcione está bien.

En cuanto a los templates simplemente he mostrado una tabla:

## Perfiles

	hector	<a href="#">Ver perfil</a>
	juanito	<a href="#">Ver perfil</a>
	HOLA	<a href="#">Ver perfil</a>

He ido generando las filas \<tr> en un bucle para cada profile en profile\_list y les he añadido una columna con el enlace:

```


|                                                                                                                                                                                                                                         |                                                                                             |                                                                   |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| {% if profile.avatar %}  {% else %}  {% endif %} | {{profile.user}} <td><a href="{% url 'profiles:detail' profile.user %}">Ver perfil</a></td> | <a href="{% url 'profiles:detail' profile.user %}">Ver perfil</a> |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|-------------------------------------------------------------------|


```

Para los perfiles públicos he reutilizado la estructura a dos columnas del formulario:



## hector

Me llamo Héctor y me gusta programar!

<https://www.hektorprofe.net/>

Como véis utilizando CBV la parte del backend es fácil, lo que más trabajo da es adaptar los templates.

Por cierto, os pedí crear por lo menos 4 usuarios de prueba, lo he hecho con la excusa de añadir paginación a nuestra ListView, trataremos el concepto en la siguiente lección.

## Paginación de resultados en ListView

A no ser que sobreescribamos el método `get_queryset` de una ListView para filtrarla (de forma similar a como sobreescrivimos el `get_object` en una `DetailView`), ésta por defecto devolverá todas las instancias.

Cuando trabajamos con poca información no pasa nada, pero imaginad si tuviéramos cientos o miles de instancias... Para estos casos existe la paginación, un sistema que permite dividir los registros en distintas páginas para navegar sin saturarnos de datos.

Sinceramente configurar la paginación en una vista tradicional es, hablando en plata, un coñazo, pero con las ListView es mega fácil. Sólo debemos definir un atributo llamado “`paginate_by`” donde estableceremos el número de registros por página:

```
class ProfileListView(ListView):
    model = Profile
    template_name = 'profiles/profile_list.html'
    paginate_by = 2
```

Si volvemos a la lista de perfiles ahora veremos que nos aparecen solo dos:

## Perfiles

	hector	<a href="#">Ver perfil</a>
	juanito	<a href="#">Ver perfil</a>

Esto es debido a que estamos mostrando la primera página del paginador. Para cargar la segunda página debemos añadir un parámetro GET llamado page con el número 2 en la url:

[127.0.0.1:8000/profiles/?page=2](http://127.0.0.1:8000/profiles/?page=2)

## Perfiles

	juanito	<a href="#">Ver perfil</a>
---	---------	----------------------------

Bueno, pues así es como gestiona automáticamente nuestra ListView la paginación. Sólo debemos maquetar un menú de paginación en la parte inferior ayudándonos de los Template Tags del paginador. Esto sí que es un poco pesado, por eso os lo voy a dar hecho, podéis copiar el que tengo en mi Gist. Lo he creado tomando como base la paginación de Bootstrap:

<https://gist.github.com/hcosta/1c29ac407b6d23afab505ab8cb1ccb36>

```
<!-- Menú de paginación -->
{% if is_paginated %}
    ...
{% endif %}
```

## Perfiles

	hector	<a href="#">Ver perfil</a>
	juanito	<a href="#">Ver perfil</a>
<span style="border: 1px solid #ccc; padding: 2px;">«</span> <span style="border: 1px solid #0070C0; color: #0070C0; padding: 2px;">1</span> <span style="border: 1px solid #ccc; padding: 2px;">2</span> <span style="border: 1px solid #ccc; padding: 2px;">»</span>		

Empezando por arriba, el Tag `is_paginated` nos sirve para saber si debemos mostrar la paginación. Si por ejemplo paginásemos a 10 resultados no nos aparecería porque sólo tenemos 4 perfiles.

Ya en el menú, todo lo referente al paginador se gestiona en un objeto llamado `page_obj` que podemos consultar en todo momento. Este incluye atributos `has_previous` y `has_next` para comprobar si la página actual tiene una página anterior y siguiente. Si es que sí nos permite recuperarlas con `previous_page_number` y `next_page_number`. Respecto al bucle de páginas, lo podemos conseguir iterando con un `for` el atributo `page_range`. Ya como añadido podemos consultar `number` para comprobar la página actual y determinar si añadir o no una clase activa al número. Sólo hay que jugar un poco con todos estos elementos para maquetar un paginador la mar de interesante.

Por cierto, no sé si os habréis fijado pero tenemos un warning al usar el paginador:

### UnorderedObjectListWarning: Pagination may yield

Esto es debido a que no tenemos una ordenación por defecto para nuestros perfiles. Solucionarlo es tan fácil como volver a `registration/models` y añadir un campo `Meta order`:

```
class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    avatar = models.ImageField(upload_to=custom_upload_to, null=True, blank=True)
    bio = models.TextField(null=True, blank=True)
    link = models.URLField(max_length=200, null=True, blank=True)

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

Si queréis más información os dejo enlaces a la documentación oficial, tanto de Django como de Bootstrap:

\ <https://docs.djangoproject.com/en/dev/topics/pagination/> \ <https://getbootstrap.com/docs/4.0/components/pagination/>

# Décima App [Messenger] Análisis y diseño

He pensado mucho en una app que nos permita no sólo poner a prueba lo que sabemos, sino también que aproveche la estructura que hemos desarrollado y además podamos aprender cosas nuevas.

Al final se me ha ocurrido algo interesante: una app de mensajería privada similar a la de Udemy. De hecho no sé si lo he comentado ya, pero Udemy está en gran parte programado con Django, así que la he utilizado de referencia.

Así que vamos a ponernos en situación analizando la funcionalidad mientras aplicamos el principio de las Cinco W (Quién - Who, Qué - What, Cuándo - When, Dónde - Where y Porqué - Why). Me gusta hacerlo porque arroja mucha luz independientemente del campo dónde se aplique, además sirve para filtrar.

- ¿Quién hará uso de la funcionalidad? Un usuario registrados e identificado.
- ¿Qué hará la funcionalidad? Establecer un chat privado entre ambos para que se comuniquen.
- ¿Cuándo? Cuando un usuario decida iniciar una conversación con otro, o un usuario visite la sección “Mensajes” para revisar los chats que tiene abiertos y continuar sus conversaciones.
- ¿Dónde? En su debida sección de Mensajes o a través del botón “Enviar mensaje” en los perfiles públicos.
- ¿Por qué? Porque ofrecer una vía de comunicación privada es una opción que toda aplicación social debería incluir, además será reutilizable y nos servirá para aprender mucho.

Cabe señalar que nuestro sistema de mensajes no será en tiempo real, es decir, no será un chat. Para eso hace falta crear websockets y se requiere un nivel más avanzado. Lo nuestro será un sistema mucho más simple, de enviar el mensaje

Ya que vamos a enviar mensajes, podríamos considerar que un Mensaje “message” es un modelo. Así que como primera idea podríamos pensar en un mensaje como un modelo con los siguientes campos:

- Un emisor: el usuario que envía el mensaje
- Un receptor: el usuario que recibe el mensaje
- Un cuerpo: con el texto del mensaje
- Una fecha: con el momento de creación

Con esto ya tenemos los mensajes, pero necesitamos una forma de manejarlos en conjunto, y ahí es donde aparece el concepto de hilo de conversación, en inglés "thread".

Podemos entender un hilo como un lugar donde ocurre la conversación, una especie de punto de encuentro entre varios usuarios donde se almacenan mensajes, por lo que ya no necesitamos un receptor en el mensaje, el receptor será el propio hilo que los contendrá. En consecuencia nos quedarían dos modelos:

- Message (mensaje): con tres campos: una relación FK al usuario que lo crea, el cuerpo y la fecha.
- Thread (hilo): con dos relaciones M2M: una con los usuarios que forman parte del hilo y otra con los mensajes que se van añadiendo al mismo.

Lo bueno de este sistema es que un hilo no tiene porqué limitarse a dos usuarios, aunque en nuestro caso así será porque es una restricción impuesta por la propia funcionalidad.

Así que vamos a dejar preparada la app y sus modelos.

Vamos a llamarla Messenger porque Django ya incluye una app Messages y no podemos repetir nombres. Como Messenger significa mensajero me pareció un buen nombre alternativo, también así que vamos a utilizar ese:

```
python manage.py startapp messenger
```

```
'core',
→ 'messenger',
```

Sin más preámbulos vamos a por nuestros modelos:

```

from django.db import models
from django.contrib.auth.models import User


class Thread(models.Model):
    users = models.ManyToManyField(User, related_name='threads')

class Message(models.Model):
    thread = models.ForeignKey(Thread, on_delete=models.CASCADE)
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    created = models.DateTimeField(auto_now_add=True)

class Message(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    content = models.TextField()
    created = models.DateTimeField(auto_now_add=True)

    class Meta:
        ordering = ['created']

class Thread(models.Model):
    users = models.ManyToManyField(User, related_name='threads')
    messages = models.ManyToManyField(Message)

```

Como véis esta parte no es muy compleja, ahora a ver cómo nos las apañamos para manejarlo.

```

python manage.py makemigrations messenger
\ python manage.py migrate messenger

```

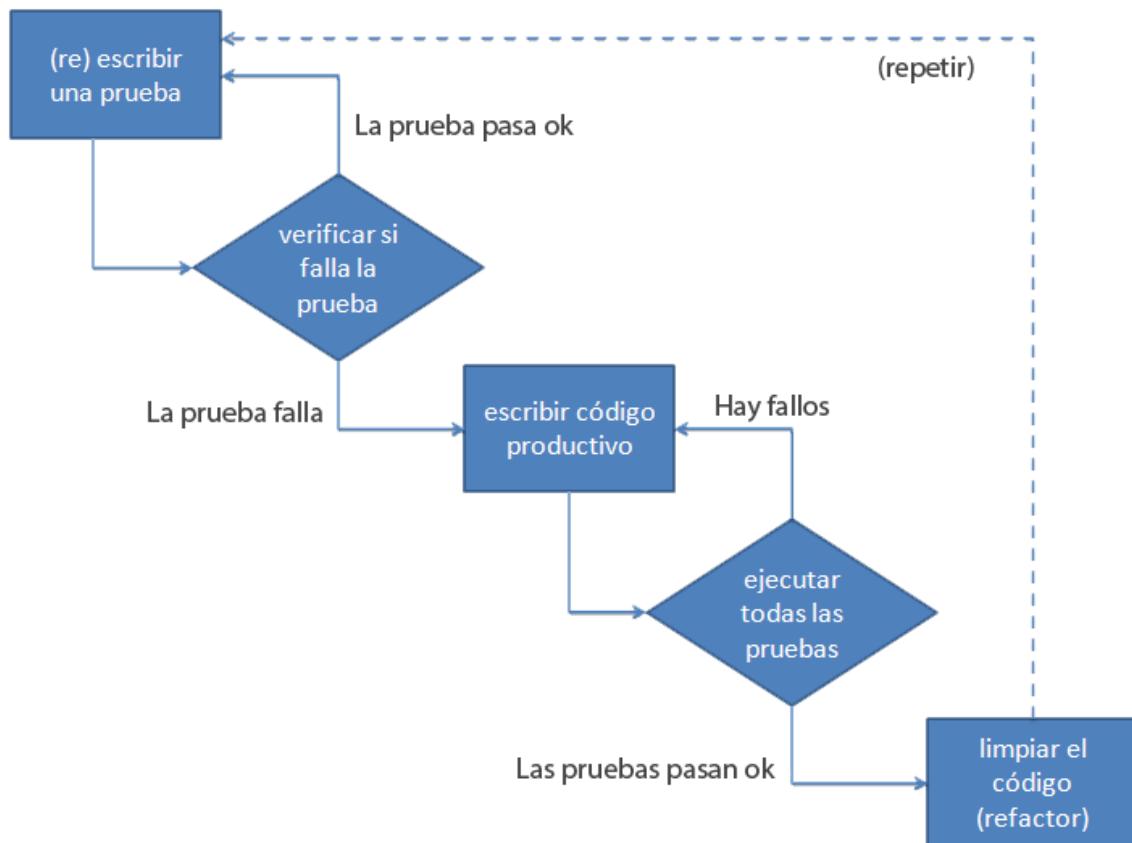
## TDD (1): Primeras pruebas

Ahora que tenemos la app con los modelos es muy tentador lanzarnos a la piscina, pero no sería buena idea. A diferencia de todo lo que hemos hecho en esta ocasión nuestro objetivo es desarrollar una funcionalidad poco común, así que necesitamos experimentar un poco antes.

La mejor forma de experimentar es hacer pruebas porque como sabéis se ejecutan en una base de datos aislada. Así que en lugar de ir directamente a por las vistas, vamos a desarrollar algunas funcionalidades básicas

relacionadas con nuestros modelos Thread y Message mientras las probamos en pequeños Tests.

Para hacerlo vamos a adoptar una práctica que en ingeniería de software se conoce como TDD, Test Driven Development (Desarrollo Guiado por Pruebas). Es una forma de programar muy interesante que se basa en crear primero la prueba y luego desarrollar la funcionalidad. Cuando el TDD se hace bien las pruebas te llevan de la mano y desvelan nuevas formas de hacer las cosas, ya lo veréis.



Cuando no tengamos muy claro por dónde empezar, siempre podemos simular situaciones básicas, como la creación de y consulta de los modelos, así que vamos a hacerlo:

```
from django.test import TestCase
from django.contrib.auth.models import User
from .models import Thread, Message

# Create your tests here.
class ThreadTestCase(TestCase):
    def setUp(self):
        self.u1 = User.objects.create_user('u1', None, 'test1234')
        self.u2 = User.objects.create_user('u2', None, 'test1234')

        self.thread = Thread.objects.create()
        self.thread.users.add(self.u1, self.u2)
```

Al crear los usuarios y el hilo como variables de instancia con `self`, podemos hacer uso de ellos en cualquier método de la clase, por ejemplo en un `test_add_users_to_thread`:

```
def test_add_users_to_thread(self):
    self.thread.users.add(self.user1, self.user2)
```

Añadir valores a un campo Many2Many es tan fácil como utilizar su método `add` y pasarlo separados por comas. Pero ésto no podemos dejarlo así, nos falta comprobar si se han añadido correctamente, así que podemos hacer una simple aserción y comparar si el hilo tiene dos usuarios:

```
def test_add_users_to_thread(self):
    self.thread.users.add(self.user1, self.user2)
    self.assertEqual(len(self.thread.users.all()), 2)
```

Ahora podemos ejecutar el tests de diferentes formas. Por ejemplo, si queremos ejecutar todos los Tests dentro del fichero `tests.py` de la app `messenger` haríamos:

```
python manage.py test messenger
```

Si en cambio queremos ejecutar sólo los de la clase `ThreadTestCase` haríamos:

```
python manage.py test messenger.tests.ThreadTestCase
```

Y si sólo quisiéramos ejecutar el test que acabamos de crear haríamos:

```
python manage.py test messenger.tests.ThreadTestCase.test_add_users_to_thread
```

Esto es útil cuando tenemos muchas Clases y tests diferentes, pero en nuestro caso tendremos poquitos, así que utilizaremos siempre la primera forma.

Sea como sea el Test valida correctamente:

```
Ran 1 test in 0.113s
```

```
OK
```

Por lo que sabemos que en efecto, esta forma de añadir Usuarios a un Hilo funciona.

¿Qué os parece ahora crear un test para recuperar un hilo ya existente a partir de sus usuarios? La forma más fácil de filtrar por dos usuarios, es simplemente anidar dos filtros de la siguiente forma:

```
def test_filter_threads_by_users(self):
    self.thread.users.add(self.user1, self.user2)
    threads = Thread.objects.filter(users=self.user1).filter(users=self.user2)
```

Ahora podemos hacer una aserción que compruebe si el Hilo donde hemos añadido los usuarios, el que es de instancia, es el mismo al que en teoría hay en la primera posición del QuerySet que devuelve filter:

```
self.assertEqual(self.thread, threads[0])
```

```
Ran 2 tests in 0.227s
```

```
OK
```

Pues parece que sí, que de esta forma se puede recuperar el hilo. Por cierto, no sé si os habéis percatado, pero para cada Test (método que empieza por test\_) se crea de nuevo la base de datos y se ejecuta el SetUp, esa es la razón por la que se nos muestra varias veces el mensaje...

```
# print("Se acaba de crear un usuario y su perfil enlazado")
```

Deberíamos comentar esta línea del modelo Profile de “registration” para que no moleste.

A todo esto, ¿y si intentamos recuperar un hilo que no existe qué pasaría? ¿El QuerySet que devuelve filter debería estar vacío no? Vamos a probarlo:

```
def test_filter_non_existent_thread(self):
    threads = Thread.objects.filter(users=self.user1).filter(users=self.user2)
    self.assertEqual(len(threads), 0)
```

Ran 3 tests in 0.365s

OK

¿Y qué hay de los mensajes? Vamos a trabajar un poco con ellos. Podemos crearlos y añadirlos prácticamente igual que los usuarios:

```
def test_add_messages_to_thread(self):
    self.thread.users.add(self.user1, self.user2)
    msg1 = Message.objects.create(user=self.user1, content="Muy buenas")
    msg2 = Message.objects.create(user=self.user2, content="Hola")
    self.thread.messages.add(msg1, msg2)
    self.assertEqual(len(self.thread.messages.all()), 2)
```

Ran 4 tests in 0.459s

OK

Si queremos hasta podemos mostrar la conversación:

```
for msg in self.thread.messages.all():
    print("({}): {}".format(msg.user, msg.content))
```
(user1): Muy buenas
(user2): Hola
```

Con estos 4 tests hemos experimentado las acciones esenciales:

- Crear hilos
- Asignarles usuarios
- Recuperar hilos a partir de sus usuarios
- Crear mensajes, asignarlos a hilos y recuperarlos

¿Puede considerarse esto TDD? Pues todavía no, porque al utilizar un framework como Django todo lo que hemos hecho hasta ahora nos viene dado. Sin embargo en las siguientes dos lecciones vamos a mejorar estas funcionalidades, y ahí si haremos TDD propiamente dicho, creando código y refactorizando hasta conseguir validar algunos Tests.

## TDD (2): Refactorización

Quizá os ha pasado por alto, o quizás no, pero tenemos un fallo de seguridad importante en el momento de añadir un mensaje a un Hilo. Os lo voy a demostrar con un Test.

Empezaremos añadiendo un tercer usuario:

```
self.user3 = User.objects.create_user('user3', None, 'test1234')
```

Ahora crearemos un test que intente añadir un tercer mensaje al hilo creado por user3 pese a que él no forme parte del hilo. Se supone que si un usuario no forma parte del hilo no debe ser capaz de añadir un mensaje, así que vamos a suponer que no, y por lo tanto éste contendrá sólo los dos primeros mensajes:

```
def test_add_message_from_user_not_in_thread(self):
    self.thread.users.add(self.user1, self.user2)
    msg1 = Message.objects.create(user=self.user1, content="Muy buenas")
    msg2 = Message.objects.create(user=self.user2, content="Hola")
    msg3 = Message.objects.create(user=self.user3, content="Soy un espía")
    self.thread.messages.add(msg1, msg2, msg3)
    self.assertEqual(len(self.thread.messages.all()), 2)
```

Como podéis suponer, al ejecutar el test nos fallará:

```
AssertionError: 3 != 2
-----
Ran 5 tests in 0.797s
FAILED (failures=1)
```

Estaba claro como el agua que se añadiría, porque en ningún momento hemos comprobado lo contrario. Tenemos que refactorizar para solucionarlo.

¿Cómo vamos a modificar el comportamiento del campo Many2ManyField para que compruebe que los mensajes los han creado usuarios que forman parte del hilo? Pues con algo que ya conocemos: una señal, concretamente una señal llamada m2m-changed, os dejaré documentación al respecto en los recursos de la lección: <https://docs.djangoproject.com/en/dev/ref/signals/#m2m-changed>

Esta señal permite detectar cuando un campo Many2Many es modificado, así que siguiendo el ejemplo de la documentación podríamos hacer lo siguiente:

```
from django.db.models.signals import m2m_changed
```

Y abajo del todo:

```
def messages_changed(sender, **kwargs):
    # Do something
    pass

m2m_changed.connect(messages_changed, sender=Thread.messages.through)
```

Como véis esta forma de crear la señal es directa y no requiere el uso de un decorador @receiver.

En todo caso, no sabemos muy bien qué nos vamos a encontrar, así que vamos a hacer un debug comprobando los argumentos instance, action y pk\_set. Según la documentación, instance hace referencia al Thread desde donde se llama la señal, action puede contener pre\_add o post\_add, indicando el momento justo antes o después de guardar los registros. Luego tenemos pk\_set, un conjunto donde se pasan las PK de los mensajes que se añadirán o han sido añadidos, dependiendo del momento:

```
def messages_changed(sender, **kwargs):
    instance = kwargs.pop("instance", None)
    action = kwargs.pop("action", None)
    pk_set = kwargs.pop("pk_set", None)
    print(instance, action, pk_set)
```

Vamos a ejecutar sólo el test actual para mostrar estos valores al llamarla la señal:

```
python manage.py test messenger.tests.ThreadTestCase.test_add_message_from_user_not_in_thread
  Thread object (1) pre_add {1, 2, 3}
  \ Thread object (1) post_add {1, 2, 3}
```

Así se ve cómo efectivamente la señal capta dos momentos en el Thread 1, pre\_add y post\_add con las PK de los mensajes.

A nosotros nos interesa el pre\_add, porque en él comprobaremos si algún mensaje tiene un usuario que no forma parte del Hilo:

```
def messages_changed(sender, **kwargs):
    instance = kwargs.pop("instance", None)
    action = kwargs.pop("action", None)
    pk_set = kwargs.pop("pk_set", None)

    if action is "pre_add":
        for msg_pk in pk_set:
            msg = Message.objects.get(pk=msg_pk)
            if msg.user not in instance.users.all():
                print("Ups... {} no forma parte del hilo".format(msg.user))

```

\ Ups... (user3) no forma parte del hilo

Ahora sólo tenemos que sacar el mensaje de este usuario del conjunto pk\_set. Sin embargo no podemos hacerlo dentro de la iteración, ya que afectaría al conjunto y quedaría inconsistente:

```
if msg.user not in instance.users.all():
    print("Ups... {} no forma parte del hilo".format(msg.user))
    pk_set.remove(msg_pk) XFAIL

```

\ RuntimeError: Set changed size during iteration

En su lugar podemos simplemente almacenar los mensajes fraudulentos en otro conjunto:

```
false_pk_set = set() # conjunto que almacena los mensajes fraudulentos
```

Los vamos agregando ahí a medida que los encontramos:

```
if msg.user not in instance.users.all():
    print("Ups... {} no forma parte del hilo".format(msg.user))
    → false_pk_set.add(msg_pk)
```

Ya al final de todo, fuera de la iteración, hacemos podemos utilizar el método difference\_update de los conjuntos para encontrar los elementos no comunes (1, 2) entre el conjunto pk\_set y false\_pk\_set, actualizándolos en pk\_set y filtrando así los mensajes:

```

def messages_changed(sender, **kwargs):
    instance = kwargs.pop("instance", None)
    action = kwargs.pop("action", None)
    pk_set = kwargs.pop("pk_set", None)

    false_pk_set = set() # conjunto que almacena los mensajes fraudulentos

    if action is "pre_add":
        for msg_pk in pk_set:
            msg = Message.objects.get(pk=msg_pk)
            if msg.user not in instance.users.all():
                print("Ups... {} no forma parte del hilo".format(msg.user))
                false_pk_set.add(msg_pk)

    # Buscamos los mensajes de false_pk_set que no están en pk_set y los borramos de pk_set
    pk_set.difference_update(false_pk_set)

```

Si ejecutamos el Test de nuevo, esta vez sí nos debería funcionar:

|                      |                       |
|----------------------|-----------------------|
| Ran 1 test in 0.166s | Ran 5 tests in 0.896s |
| OK                   | OK                    |

Esto sí que es TDD en toda regla, ya que a fuerza de refactorizar hemos conseguido validar un test sin alterar su estructura.

Si esta lección os ha parecido interesante, entonces la siguiente os gustará todavía más, pues añadiremos nuestro propio Model Manager para el modelo Thread, ya veréis que útil.

## TDD (3): Creando un Model Manager

Bien, pues ya sabemos cómo manejar nuestros modelos, incluso hemos solventado el fallo de que un usuario no pueda enviar mensajes a un hilo al cual no pertenece, pero sigo echando en falta algo, y eso es una forma más sencilla de recuperar un Thread a partir de sus usuarios, digamos que no me gusta lo de añadir filter dos veces. Así que ¿por qué no creamos una consulta a medida dentro de Thread para recuperar una instancia a partir de dos usuarios? Vamos a hacerlo.

Ya que hablamos de consultas a medida, lo que necesitamos es filtrar el QuerySet, y eso es algo que requiere crear nuestro propio Model Manager o Gestor de Modelos.

Al hacerlo podremos añadir un método como Thread.objects.find(user1, user2) donde a partir de dos usuarios se busque el hilo de forma fácil y cómoda.

Vamos a empezar creando un test:

```
def test_find_thread_with_custom_manager(self):
    self.thread.users.add(self.user1, self.user2)
    thread = Thread.objects.find(self.user1, self.user2)
    self.assertEqual(self.thread, thread)
```

Obviamente no va a pasar porque el método todavía no existe, pero esa es la gracia, refactorizar y pasar el test:

```
AttributeError: 'ThreadManager' object has no attribute 'find'

-----
Ran 6 tests in 1.026s

FAILED (errors=1)
```

Si queremos añadir nuestros propios filtros necesitamos crear un objects Manager, y eso lo haremos en models.py, podríamos llamarlo ThreadManager:

```
class ThreadManager(models.Manager):
    pass
```

Ahora asignaremos nuestro Manager al campo objects del modelo Thread:

```
objects = ThreadManager()
```

Con esto ya podemos centrarnos en crear el filtro. Es bien fácil, pues dentro de un ThreadManager, la variable self hace referencia al propio QuerySet, sólo tenemos que filtrarla así misma y devolver el resultado:

```
class ThreadManager(models.Manager):
    def find(self, user1, user2):
        querySet = self.filter(users=user1).filter(users=user2)
        if len(querySet) > 0:
            return querySet[0]
        return None
```

Si encontramos algún Thread lo devolvemos, y en caso contrario pues devolvemos None.

Una vez creado el método, debería funcionarnos:

```
Ran 6 tests in 2.116s

OK
```

También podríamos comprobar si devuelve None con un hilo inexistente:

```
def test_find_thread_with_custom_manager(self):
    self.thread.users.add(self.user1, self.user2)
    thread = Thread.objects.find(self.user1, self.user2)
    self.assertEqual(self.thread, thread)
    thread = Thread.objects.find(self.user1, self.user3)
    self.assertEqual(None, thread)
```

```
Ran 6 tests in 1.039s
```

```
\ OK
```

¿Mucho más fácil que andar haciendo múltiples filtros no?

¿Qué os parece si añadimos una última funcionalidad, a modo de get\_or\_create, podemos hacer un filtro find\_or\_create, que si no encuentra un Hilo lo cree automáticamente, lo cual sería la guinda del pastel y nos facilitaría muchísimo el trabajo. Vamos primero con el test:

```
def test_find_or_create_thread_with_custom_manager(self):
    self.thread.users.add(self.user1, self.user2)
    thread = Thread.objects.find_or_create(self.user1, self.user2)
    self.assertEqual(self.thread, thread)
    thread = Thread.objects.find_or_create(self.user1, self.user3)
    self.assertIsNotNone(thread)
```

Que obviamente no pasará:

```
AttributeError: 'ThreadManager' object has no attribute 'find_or_create'

-----
Ran 7 tests in 1.125s

FAILED (errors=1)
```

Vamos a crearlo:

```
def find_or_create(self, user1, user2):
    thread = self.find(user1, user2)
    if thread is None:
        thread = Thread.objects.create()
        thread.users.add(user1, user2)
    return thread
```

```
Ran 7 tests in 1.207s
```

```
\ OK
```

Con esto estamos listos para implementar las futuras vistas de nuestra app con total seguridad. El TDD nos ha llevado de la mano en nuestros experimentos y nos ha permitido desarrollar las distintas funcionalidades de las que haremos uso en las siguientes lecciones.

## Urls, vistas y templates

No sirve de nada crear unas vistas sin contenido, así que antes vamos a añadir algunos datos a nuestros modelos. Podemos hacerlo desde la shell o configurando el admin, yo como soy más de terminal utilizaré la primera forma.

```
python manage.py shell

from django.contrib.auth.models import User
\ from messenger.models import Thread, Message

hector = User.objects.get(username="hector")
\ juanito = User.objects.get(username="juanito")

thread = Thread.objects.find_or_create(hector, juanito)

>>> thread.messages.add(Message.objects.create(user=hector, content="Buenos días Juanito!"))
>>> thread.messages.add(Message.objects.create(user=juanito, content="Hola Héctor"))
>>> thread.messages.add(Message.objects.create(user=hector, content="Me voy"))
>>> thread.messages.add(Message.objects.create(user=juanito, content="Pues adiós"))

>>> exit()
```

Ahora sí, vamos con las vistas. Necesitamos por lo menos dos, una para mostrar los hilos del usuario, y otra para mostrar un hilo en concreto, por lo que podríamos crear:

- Una ListView para listar todos los hilos de un usuario en /messenger/

- Una DetailView para listar todos los mensajes de un hilo en /messenger/thread/\\<int:pk>/

```
from django.views.generic.list import ListView
from django.views.generic.detail import DetailView
from .models import Thread

# Create your views here.
class ThreadListView(ListView):
    model = Thread

class ThreadDetailView(DetailView):
    model = Thread
```

Evidentemente no podemos dejarlas así. En el caso de la ListView no queremos devolver todos los Thread, sólo los del usuario, así que debemos filtrarlos. Para conseguirlo sobreescriviremos el método get\_queryset y filtraremos por el request.user. Esto implica que el usuario esté identificado, así que necesitamos decorar la CBV con login\_required:

```
from django.contrib.auth.decorators import login_required
from django.utils.decorators import method_decorator

@method_decorator(login_required, name='dispatch')
class ThreadListView(ListView):
    model = Thread

    def get_queryset(self):
        queryset = super(ThreadListView, self).get_queryset()
        return queryset.filter(users=self.request.user)
```

La lógica normal sería esta, pero dado que nuestros usuarios tienen una relación con el modelo Thread a partir del campo Many2Many messages, podríamos acceder directamente en el template y ahorrarnos toda el ListView, usando en su lugar una simple TemplateView:

```
from django.views.generic import TemplateView
\

@method_decorator(login_required, name='dispatch')
class ThreadListView(TemplateView):
    template_name = "messenger/thread_list.html"
```

Respecto a la DetailView debemos sobreescribir el método get\_object, ahí comprobaremos si el request.user forma parte del hilo, y si no es así lanzaremos un error 404:

```
from django.http import Http404
\

@method_decorator(login_required, name='dispatch')
class ThreadDetailView(DetailView):
    model = Thread

    def get_object(self):
        obj = super(ThreadDetailView, self).get_object()
        if self.request.user not in obj.users.all():
            raise Http404()
        return obj
```

Ahora que tenemos las views vamos a configurar las URL, pondréas como yo porque os facilitaré un par de templates ya preparados y utilizarán esta lógica:

```
from django.urls import path
from .views import ThreadListView, ThreadDetailView

messenger_patterns = [
    path('', ThreadListView.as_view(), name='list'),
    path('thread/<int:pk>', ThreadDetailView.as_view(), name='detail'),
], 'messenger'

from messenger.urls import messenger_patterns
# Paths de messenger
\ path('messenger/', include(messenger_patterns)),
```

Nuestras vistas cargarán los templates thread\_list.html y thread\_detail.html, como os comentaba os los doy hechos para no alargar demasiado la lección, los podréis descargar en los recursos y copiarlos en un nuevo directorio templates/messenger en la propia app:

The screenshot shows the WinRAR interface. The file list contains:

| Nombre             | Tamaño | Tamaño comprimido |
|--------------------|--------|-------------------|
| Instrucciones.txt  | 365    | 181               |
| thread_detail.html | 3 802  | 1 035             |
| thread_list.html   | 2 349  | 690               |

Below the file list, a tree view shows the directory structure:

```

templates
  messenger
    thread_detail.html
    thread_list.html
  
```

He añadido comentarios en los Templates para que podáis analizarlos vosotros mismos, si tenéis alguna duda me lo decís.

Ya sólo nos faltaría añadir la sección message en el menú superior si estamos identificados:

```

{%
  else %}
<li class="nav-item">
| <a class="nav-link" href="{% url 'messenger:list' %}>Mensajes</a>
</li>
  
```

Y probar si todo funciona:

The application interface includes a navigation bar with links for Playground, Inicio, Páginas, Perfiles, Mensajes, Perfil, and Salir.

The main content area shows a conversation between two users:

- Mensajes con juanito**
- juanito** (Profile picture) *Hace 2 horas, 52 minutos*: Buenos días Juanito!
- Héctor** *Hace 2 horas, 52 minutos*: Hola Héctor
- juanito** *Hace 2 horas, 52 minutos*: Me voy
- Héctor** *Hace 2 horas, 52 minutos*: Pues adiós

En la siguiente lección añadiremos el formulario para enviar mensajes.

# Mensajes asíncronos con JS (1): Primer concepto

El formulario de mensajes tiene la estructura más simple del mundo, un único campo con un texto. ¿Vale la pena diseñar un form para eso? Pues no. Además es una ocasión excelente para enseñarlos a trabajar con peticiones asíncronas.

¿Qué es una petición asíncrona? Para entenderlo podemos simplemente tomar un ejemplo real. Imaginemos un usuario navegando por una página web. En un momento dado decide cambiar de sección y hace clic en un enlace. Al hacerlo, se hace una petición al servidor para que éste devuelva la nueva página.

- Si este proceso es síncrono, al hacer la petición el cliente queda en un limbo esperando la respuesta, digamos que se sincroniza con el servidor. Si el servidor tarda 5 segundos en devolver la nueva página, serán 5 segundos que el cliente estará bloqueado esperando la respuesta.
- En cambio, si este proceso es asíncrono, el cliente pide la página pero puede seguir funcionando. En otras palabras, la petición ocurre en segundo plano sin que el cliente se percate de ello, y únicamente será consciente al recibirse la respuesta.

Este tipo de peticiones son muy útiles porque no requieren recargar toda la página, sino que se puede modificar sólo una sección determinada en función de la respuesta, haciendo la experiencia mucho más fluida. Sin duda la forma ideal para enviar mensajes sin que el usuario deba recargar toda la página.

¿Pero cómo se logra este comportamiento? Pues gracias al lenguaje Javascript, que es capaz de crear un objeto en memoria donde manejar la petición, capturar la respuesta y modificar el DOM, la interfaz que permite modificar el contenido del HTML en tiempo real.

Pero basta de introducciones y vamos a entrar en materia, empezando por identificar qué datos necesitamos para crear un mensaje y cómo podemos enviarlos a una vista clásica:

- El hilo: lo pasaremos en el PATH de la petición con una variable pk
- El usuario: ya lo tenemos en la propia petición request
- El contenido: lo enviaremos en una petición GET

Podríamos enviarlo en una petición POST, pero como requiere el token csrf es más fácil hacerlo por GET.

Así que vamos a crear la vista:

```
def add_message(request, pk):
    pass
```

El respectivo PATH quedaría así:

```
from .views import ThreadListView, ThreadDetailView, add_message
path('thread/<int:pk>/add/', add_message, name='add'),
```

Si accedemos a nuestra nueva vista nos dará error porque no estamos devolviendo nada:

ValueError at /messenger/thread/1/add/

The view messenger.views.create\_message didn't return an HttpResponseRedirect object.

¿Qué tenemos que devolver en una petición asíncrona? Pues lo que queramos, puede ser texto plano, un snippet HTML para injectarlo directamente en la página, o una estructura bien organizada en formato XML o JSON que podemos analizar para actuar en consecuencia.

Mi recomendación es responder siempre con estructuras JSON, unos objetos muy parecidos a un diccionario de Python pero escritos con la sintaxis de Javascript, algo muy útil, porque como la propia petición se realiza en Javascript, la respuesta se puede manejar al vuelo sin hacer conversiones.

Para poder devolver una respuesta JSON necesitamos importar el módulo JsonResponse:

```
from django.http import JsonResponse
```

Ahora crearemos la respuesta como si fuera un diccionario Python indicando que inicialmente no se ha creado el mensaje y la transformaremos a JSON para devolverla:

```
def add_message(request, pk):
    json_response = {'created': False}
    return JsonResponse(json_response)
```

Si actualizamos la página funcionará mostrando nuestro diccionario:

```
{"created": false}
```

Sin embargo fíjate bien en el resultado, pues se trata de un objeto JSON y no un diccionario de Python. Es prácticamente igual pero cambia el booleano False con la f en minúscula porque en Javascript se escribe de esa forma.

Lo bueno es que este objeto ya es reconocible por Javascript, por lo que podemos empezar a desarrollar la parte del formulario. Luego ya volveremos a nuestra vista.

Vamos a empezar haciendo un experimento, a ver si somos capaces de hacer una petición asíncrona a la vista add\_message presionando un botón y mostrando el resultado en pantalla.

Dado que prácticamente todos los navegadores implementan las funciones de ECMA Script 6, para hacer una petición asíncrona ya no es necesario hacer malabarismos ni utilizar librerías como jQuery. Sólo debemos utilizar la función fetch, la cual toma una url para hacer la petición y también es capaz de manejar la respuesta:

Explicar línea a línea el experimento, separar fetch de .then, .then:

```
<!-- Aquí crearemos el formulario -->
<button id="testBtn">Petición asíncrona de prueba</button>
<script>
    var testBtn = document.getElementById("testBtn");
    testBtn.addEventListener("click", function(){
        const url = "{% url 'messenger:add' thread.pk %}"
        fetch(url)
            .then(response => response.json())
            .then(function(data) {
                alert(data.created)
            });
    });
</script>
```

## Petición asíncrona de prueba

De 127.0.0.1:8000

false

Aceptar

¡Perfecto! Como véis obtenemos el valor de created en una alerta.

Ahora viene lo bueno, necesitamos capturar el contenido del mensaje y enviarlo a la vista al presionar el botón.

Todo eso lo haremos en la siguiente lección que es una continuación de esta, pero para que os hagáis una idea vamos a hacer una simulación. En nuestra vista mostraremos un print de los parámetros GET:

```
def add_message(request, pk):
    print(request.GET)
    json_response = {'created': False}
    return JsonResponse(json_response)
```

Y crearemos un contenido de prueba que concatenaremos como parámetro GET a la url:

```
const content = "Esto es una prueba"
const url = "{% url 'messenger:add' thread.pk %}" + "?content=" + content;
```

Al presionar el botón nos aparecerá en el debug:

```
<QueryDict: {'content': ['Esto es una prueba']}
```

Bueno, pues como os decía, en la siguiente lección crearemos un textarea y tomaremos este contenido de ahí.

## Mensajes asíncronos con JS (2): Desarrollo

El textarea que almacenará el mensaje tendrá una estructura típica con bootstrap, pero en lugar de darle un name, le daremos un id, puesto que lo recuperaremos utilizando Javascript:

```
<!-- Aquí crearemos el formulario -->
<textarea id="content" class="form-control mb-2" rows="2" placeholder="Escribe tu mensaje aquí"></textarea>
```

Ya que estamos podemos modificar el botón y hacer que se vea mejor:

```

<button id="send" class="btn btn-primary btn-sm btn-block">Enviar mensaje</button>
<script>
  var send = document.getElementById("send");
  send.addEventListener("click", function(){
    const content = encodeURIComponent(document.getElementById("content").value);
    const url = "{% url 'messenger:add' thread.pk %}" + "?content=" + content;
    fetch(url)
      .then(response => response.json())
      .then(function(data) {
        alert(data.created)
      });
  });
</script>

```

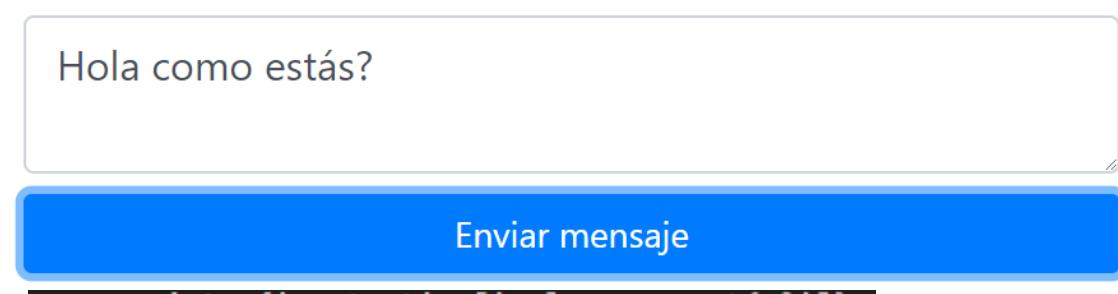
Ahora viene la parte dinámica, pues tenemos que modificar el Script para recuperar el contenido del textarea al presionar el botón, y no sólo eso, también hay que codificarlo para que no dé problemas al enviarlo por GET:

```

<button id="send" class="btn btn-primary btn-sm btn-block">Enviar mensaje</button>
<script>
  var send = document.getElementById("send");
  send.addEventListener("click", function(){
    const content = encodeURIComponent(document.getElementById("content").value);
    const url = "{% url 'messenger:add' thread.pk %}" + "?content=" + content;
    fetch(url)
      .then(response => response.json())
      .then(function(data) {
        alert(data.created)
      });
  });
</script>

```

Habiendo hecho esto si enviamos algo desde el formulario debería aparecernos en los parámetros GET de la vista:



Esto ya va tomando forma, pero faltan bastantes cosas.

Lo primero una validación que no permita enviar mensajes vacíos, así que antes de hacer el fetch() comprobaremos que el mensaje tiene por lo menos un carácter:

```

send.addEventListener("click", function(){
  const content = encodeURIComponent(document.getElementById("content").value);
  if (content.length > 0){
    const url = "{% url 'messenger:add' thread.pk %}" + "?content=" + content;
    fetch(url)
      .then(response => response.json())
      .then(function(data) {
        alert(data.created)
      });
  }
});

```

Esto quedaría aún mejor implementando un Evento con Javascript que compruebe si el textarea tiene algún valor mientras se escribe. Si lo tiene activamos el botón de envío, y si no lo desactivamos. Eso sí, inicialmente tenemos que tener el botón desactivado:

```

<button id="send" class="btn btn-primary btn-sm btn-block" disabled>Enviar mensaje</button>

/* Esto es un plus */
var content = document.getElementById("content");
content.addEventListener("keyup", function(){
  if(!this.checkValidity() || !this.value){
    send.disabled = true;
  } else {
    send.disabled = false;
  }
});

```

Ya sé que no es un curso de JavaScript, pero es que me encanta el resultado:

Escribe tu mensaje aquí

Enviar mensaje

Ahora vamos a lo importante: procesar este contenido que nos llega a la vista.

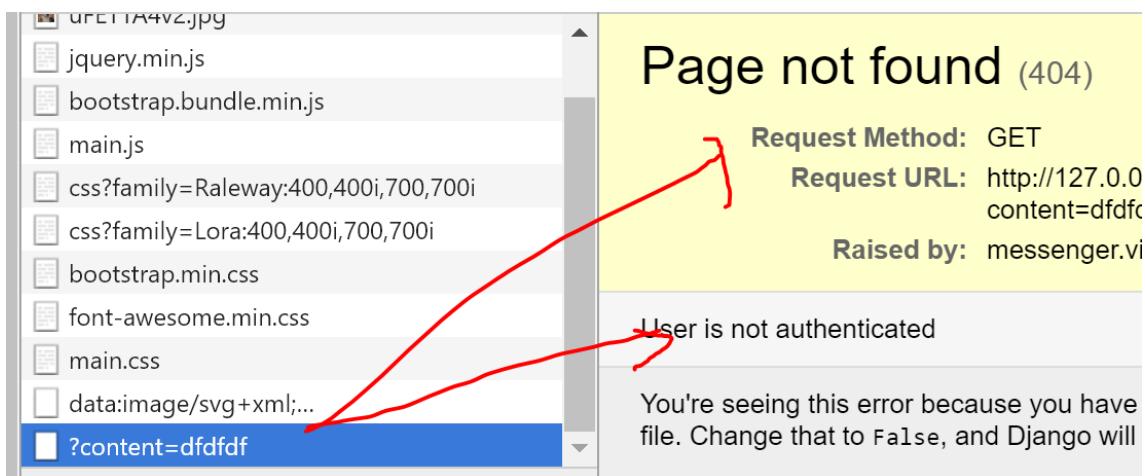
Lo primero será recuperar al usuario, por lo que necesitaremos que esté identificado. En caso de que no lo esté invocaremos un error 404:

```

def add_message(request, pk):
    json_response = {'created': False}
    if request.user.is_authenticated:
        pass
    else:
        raise Http404
    return JsonResponse(json_response)
else:
    raise Http404("User is not authenticated")
return JsonResponse(json_response)

```

Lo curioso en este momento, es que si intentamos enviar un mensaje nos va a dar error:



¿Cómo es esto posible? Si se supone que estamos identificados...

Bueno, esto ocurre porque una petición fetch() no envía las credenciales del usuario identificado, para incluirlas en la petición debemos indicarlo explícitamente:

```
fetch(url, {"credentials": 'include'})
```

Habiendo hecho esto la sesión actual se enviará también en la petición asíncrona y ahora sí debería detectar correctamente si el usuario está identificado para volver a darnos false:

De 127.0.0.1:8000

false

Aceptar

Respecto al backend ya sólo nos falta recuperar el Contenido y el Hilo para poder crear el mensaje si hay algún mensaje. Además una vez lo creemos modificaremos la clave created y la pondremos a True dando a entender que todo ha ido bien:

```
from django.shortcuts import get_object_or_404
def add_message(request, pk):
    json_response = {'created': False}
    if request.user.is_authenticated:
        content = request.GET.get('content', None)
        if content:
            thread = get_object_or_404(Thread, pk=pk)
            message = Message.objects.create(user=request.user, content=content)
            thread.messages.add(message) ←
            json_response['created'] = True
    else:
        raise Http404
    return JsonResponse(json_response)
```

Por fin ha llegado el momento de la verdad, ¿funcionará nuestro código? Vamos a probarlo:

Buenos días Juanito!

Enviar mensaje

Por lo pronto devuelve True, así que vamos a actualizar la página con F5 a ver si aparece:

De 127.0.0.1:8000

true

Aceptar

Y ahí lo tenemos, abajo del todo está el nuevo mensaje:

*Hace 0 minutos*

Buenos días Juanito!

Ya sólo nos falta desarrollar el Javascript para mostrar el mensaje sin recargar la página directamente y algún que otro detalle, pero en principio el backend lo tenemos listo.

Así que vamos a tomarnos un descanso y afinamos los detalles en la tercera parte de esta lección.

## Mensajes asíncronos con JS (3): Detalles

El primer detalle que podéis observar es que al acceder al Hilo nos aparece un Scroll vertical arriba del todo. Lo ideal sería que el Scroll siempre estuviera abajo, justo en el último mensaje. Vamos a forzarlo con Javascript en un plis plas.

La capa que tiene el Scroll es la del thread, vamos a darle una ID para poder recuperarla y forzar ese scroll a la parte de abajo dando una altura igual al propio scroll:

```
<div class="thread" id="thread">  
    // Forzamos el Scroll abajo del todo  
    function scrollBottomThread(){  
        var thread = document.getElementById("thread");  
        thread.scrollTop = thread.scrollHeight;  
    }  
  
    scrollBottomThread();  
\
```

Hace 7 minutos

Pues adiós

Hace 7 minutos

fffgfgfg

Hace 7 minutos

Buenos días Juanito!

Escribe tu mensaje aquí



Bien, y ahora vamos a lo importante. Hay dos cosas que debemos implementar.

La primera es que al enviar un mensaje en lugar de mostrarnos el True por pantalla se añada a la conversación sin necesidad de actualizar la página.

Esto es un poco tedioso, requiere crear dinámicamente una capa con el mensaje, añadirle las clases y el contenido. Luego insertarla al final de la lista de mensajes y hacer el scroll hasta abajo:

```
.then(function(data) {
  if (data.created) {
    var message = document.createElement('div');
    message.classList.add("mine", "mb-3");
    message.innerHTML = '<small><i>Hace unos segundos</i></small><br>' + decodeURIComponent(content);
    document.getElementById("thread").appendChild(message);
    scrollBottomThread(); // Hacemos el scroll abajo
  }
});
```

Visto en perspectiva en el desarrollo web hay un montón de trabajo, pero es muy reconfortante ver el resultado final funcionando:

*Hace unos segundos*

El resultado final es reconfortante :-}

El resultado final es reconfortante :-}

Por cierto, ¿véis que al enviar el mensaje nos sigue apareciendo en el textarea? Eso es lo segundo que nos falta, borrarlo:

```
if (content.length > 0){
  document.getElementById("content").value = ''; // Borramos el contenido
```

Y con esto ya lo tenemos. Por cierto, podríamos probar a acceder con el otro usuario a ver como se ve la conversación desde el otro lado ¿no?



Se ve perfecto y funciona todavía mejor.

Estamos muy cerca de terminar el proyecto, sólo nos falta añadir un botón para iniciar conversaciones desde el perfil de otros usuarios. Lo hacemos en la siguiente lección.

## Iniciando las conversaciones

Pues sí, hemos llegado a la penúltima lección de este mega proyecto, donde añadiremos la funcionalidad de iniciar conversaciones. Vamos a por ello.

Cuando presionemos el botón de enviar un mensaje desde un perfil, lo que haremos es una petición a una vista que creará el hilo para seguidamente redireccionarnos al hilo para empezar la conversación. Se hace rápido:

```
from django.contrib.auth.models import User
\

@login_required
def start_thread(request, username):
    user = get_object_or_404(User, username=username)
    thread = Thread.objects.find_or_create(user, request.user)
    return redirect(reverse('messenger:detail', args=[thread.pk]))
```

```
from .views import ThreadListView, ThreadDetailView, add_message, start_thread  
\  
path('thread/start/<username>', start_thread, name='start'),
```

Finalmente sólo falta añadir el respectivo botón en el template profile\_detail cuando el usuario sea distinto

```
<div class="col-md-2">  
    {% if profile.avatar %}  
          
    {% else %}  
          
    {% endif %}  
    {% if request.user != profile.user %}  
        <a href="{% url 'messenger:start' profile.user.username %}" class="btn btn-primary btn-sm btn-block mt-3">Enviar mensaje</a>  
    {% endif %}  
</div>
```

Y si todo ha ido bien, ya lo tendremos:



Enviar mensaje

Claro, hasta que no enviamos un mensaje y recarguemos no nos aparecerá la conversación a la izquierda. Se podría detectar con javascript y si es el primer mensaje actualizar la propia página:

```
json_response = {'created': False, 'first': False}  
json_response['created'] = True  
if len(thread.messages.all()) is 1:  
    json_response['first'] = True
```

```

scrollBottomThread(); // Hacemos el scroll abajo

// Si es el primer mensaje del hilo actualizamos para que aparezca a la izquierda
if (data.first){
    window.location.href= "{% url 'messenger:detail' thread.pk %}";
}

```

The screenshot shows a messaging application window titled "Mensajes con hector". On the left, there is a profile picture of a person with glasses and the name "hector" next to it. Below the name is the text "Hace 0 minutos". A red arrow points from the text "Hace 0 minutos" to the code snippet above. The main message area contains a message from "hector" with the timestamp "Hace 0 minutos" and the content "Hola Hector, solo decirte que me encanta tu curso...". Below the message area is a text input field with the placeholder "Escribe tu mensaje aquí" and a blue "Enviar mensaje" button.

## El detalle final

Si habéis experimentado un poco os habréis dado cuenta de un detalle muy importante que falta, y ese es que los hilos salgan ordenados de más recientes a más antiguos.

¿Se os ocurre alguna forma de hacerlo? Haber si sois capaces de en muy pocos pasos arreglar este problema, os aseguro que sólo hay que añadir 4 líneas en nuestro fichero models.py.

Para solucionar este problema debemos crear un campo updated en nuestro modelo Thread. Este detectará cuando se modificó por última vez y nos permitirá añadir un ordenamiento automático en la pertinente clase Meta.

```

updated = models.DateTimeField(auto_now=True)

class Meta:
    ordering = [ '-updated' ]

```

Sin embargo cuando añadimos modificamos un campo many2many updated no se actualiza, y aquí el mayor truco de este ejercicio, simular un guardado para actualizar el campo updated. ¿Dónde? ¡Pues en la señal que valida los mensajes!

```

# Buscamos los mensajes de false_pk_set que no están en pk_set y los borramos de pk_set
pk_set.difference_update(false_pk_set)

instance.save() # El truco para actualizar update consiste en guardar de nuevo la instancia

```

Haciendo hecho esto, migramos:

```
python manage.py makemigrations messenger  
\\ python manage.py migrate messenger
```

Y a partir de este momento, cuando se añada un mensaje a un Hilo, este aparecerá arriba del todo al recargar la página.

The screenshot shows a messaging application interface titled "Mensajes con juanito". On the left, there is a list of messages from three users: "juanito" (with a dog profile picture), "HOLA" (with a person icon profile picture), and "hector" (with a person icon profile picture). Each message includes the recipient's name, the time it was sent ("Hace 0 minutos"), and the message content. The message from "juanito" is highlighted with a blue background and the text "Tu eres el último". To the right of the message list is a text input field with the placeholder "Escribe tu mensaje aquí" and a blue "Enviar mensaje" button at the bottom.

Como siempre Django haciendo fácil lo más difícil.

Con esto estimados alumnos, acabamos el proyecto, espero que hayáis aprendido muchísimo y podáis aplicar muchas de las funcionalidades vistas en vuestros futuros proyectos.