Notas Django - ref documentación

### **1. Vista Detalles de Objetos(VBF): get\_object\_or\_404() vs. .get()**

En las clases, para obtener un producto específico, usamos  
 p = producto.objects.get(pk=pk). En este proyecto, para las vistas de detalle (por je de un Articulo), estamos usando get\_object\_or\_404().

* **producto.objects.get(pk=pk):**
  + **¿Qué hace?** Intenta recuperar un único objeto que coincida con el criterio.
  + **Si no lo encuentra:** Lanza una excepción DoesNotExist, lo que resulta en un **Error 500 (Error del Servidor)** si no se maneja explícitamente con un try-except. Esto no es ideal para el usuario.
* **get\_object\_or\_404(Modelo, pk=pk) (o con slug=slug):**
  + **¿Qué hace?** importa un django.shortcuts que intenta recuperar el objeto.
  + **Si no lo encuentra:** Automáticamente lanza una excepción Http404. Django convierte esto en una **página de "404 No Encontrado"** amigable para el usuario.
* **Documentación:**
  + **get\_object\_or\_404():**  [https://docs.djangoproject.com/en/5.2/topics/http/shortcuts/#get-object-or-404](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/topics/http/shortcuts/%23get-object-or-404)
  + **Consultas a la base de datos (.get() y excepciones):** [https://docs.djangoproject.com/en/5.2/topics/db/queries/#retrieving-single-objects](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/topics/db/queries/%23retrieving-single-objects)

### **2 Contexto de Plantillas y Pasando Datos: Uasndo contexto**

en blog

def lista\_articulos(request):

articulos = articulo.objects.all().order\_by('-creado') # \*\*\* nuevos primero

contexto = {'articulos': articulos,}

return render(request, 'articulos/articulos.html', contexto)

en clase

def Detalle\_producto(request, pk):

p = producto.objects.get(pk=pk)

return render (request, 'productos/Detalle\_producto.html ', {'producto': p} )

en clase

Usar variables de py en el html

**Documentación:**

* **render() y Contexto:** [https://docs.djangoproject.com/en/5.2/topics/http/shortcuts/#render](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/topics/http/shortcuts/%23render) (context)
* **Plantillas de Django (**variables**):**<https://docs.djangoproject.com/en/5.2/ref/templates/language/#variables>
* [https://docs.djangoproject.com/en/5.0/ref/templates/api/#writing-your-own-context-processors](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/ref/templates/api/%23writing-your-own-context-processors)

### **3. URLs con Slugs (SlugField y Convertidores de Ruta)**

En lugar de los IDs - pk ('Detalle/<int:pk>') que usamos en clase para los detalles de los prod , estamos usando "slugs" (ej., /articulos/mi-primer-articulo/).

* **SlugField en models.py:**
  + Se genera automáticamente a partir del título
  + **Convertidores de Ruta:**En articulos/urls.py, usamos path('<slug:articulo\_slug>/', ...) para capturar la parte del slug de la URL.  
    Django entiende que el segmento es un slug y lo pasa a la vista como un argumento nombrado (articulo\_slug).
* Nos da URLs más legibles, y mejores para el SEO.
* **Documentación:**
  + **SlugField:**https://docs.djangoproject.com/en/5.2/topics/db/models/#:~:text=fields%3A%3D%20kwargs.%20get%28%22update\_fields%22%29
  + **Convertidores de Ruta (path()):**<https://docs.djangoproject.com/en/5.2/topics/http/urls/#:~:text=int%3Apk%26gt%3B/%22%2C%20views.%20DetailView.%20as_view%28%29%2C%20name%20%3D%20%22detail%22%29%2C> (slug/pk)

### **4. Llamadas Django en html, para plantillas con for**

**{% for articulo in page\_obj.object\_list %}**: Este es un **bucle for** de Django.

* page\_obj.object\_list es la lista de objetos Articulo que tu vista lista\_articulos le pasó a la plantilla para la página actual.
* Por cada articulo en esa lista, todo el código HTML dentro del {% for %} y {% endfor %} se repite.

**{{ articulo.titulo }}**: Esto es una **variable de plantilla** de Django. Accede al campo titulo del objeto articulo actual en el bucle y lo imprime en el HTML. Lo mismo ocurre con articulo.imagen.url, articulo.cuerpo, articulo.creado, articulo.categoria.nombre.

**{% if articulo.imagen %} / {% endif %}**: Esto es una **sentencia condicional** de Django. Solo mostrará el bloque HTML de la imagen si el artículo tiene una imagen asociada.

{{articulo.resumen}} en lugar del trunc

**|date:"d M Y"**: Otro filtro que formatea la fecha del campo creado.

**{% url 'detalle\_articulo' articulo.slug %}**: Este es un **tag de URL** de Django. Genera una URL dinámica.

* 'detalle\_articulo' es el name que le diste a la URL en tu urls.py para la vista de detalle.
* articulo.slug es el valor que se le pasa a esa URL para identificar el artículo específico (asumiendo que tu URL de detalle usa el slug, como path('<slug:articulo\_slug>/', views.detalle\_articulo, name='detalle\_articulo')).

**{% empty %}**: Es una característica del bucle for que permite mostrar un contenido alternativo si la lista sobre la que se itera está vacía. (sino muestra el template vacio)

**Lógica de Paginación (page\_obj.has\_previous, page\_obj.previous\_page\_number, page\_obj.number, page\_obj.paginator.num\_pages, page\_obj.paginator.page\_range, etc.)**: Estas son propiedades del objeto page\_obj que te permiten construir los enlaces para ir a la página anterior, siguiente, y la lista de números de página.

paginator Documentacion https://docs.djangoproject.com/en/stable/topics/pagination

**request.GET.get('page')**: request.GET es un diccionario que contiene los parámetros de la URL (después del ?). .get('page') intenta obtener el valor asociado a la clave 'page' (ej. ?page=2). Si no existe, devuelve None.

### **5. Imágenes Obligatorias**

* Articulo imagen **obligatorio**. models.py con blank=False (y null=False)

**Categorías:**

**{% if articulo.categoria %}** (si no tiene categoría solo se va a mostrar Cultura y si tiene va a mostrar la cat de la base de datos) siempre 1 tag filtro minimo

**Bucle FOR para articulos**

[**https://docs.djangoproject.com/en/5.2/ref/templates/builtins/**](https://docs.djangoproject.com/en/5.2/ref/templates/builtins/)

**page\_obj.object\_list**: page\_obj es el objeto Page de la view y object\_list es la lista de los artículos por pag (12 definidos)

Dentro del bucle, se accede a los atributos de cada articulo usando la sintaxis {{ articulo.nombre\_del\_campo }}

{{ articulo.titulo }}: Muestra el título del artículo

https://docs.djangoproject.com/en/5.2/topics/pagination/

### **6. richtextfield con CKeditor - para poder darle formato al texto y agregar imagenes (entre otras cosas) a cada artículo y no solo lo de la portada**

Rich text para admin y para la pag crear   
  
<https://www.geeksforgeeks.org/python/richtextfield-django-models/>  
<https://django-ckeditor.readthedocs.io/en/latest/>  
<https://medium.com/@yashnarsamiyev2/how-to-add-ckeditor-in-django-aa6de5a09862>

### **7. Modelos (Django Models)**

* **Fundamentos de modelos:**<https://docs.djangoproject.com/en/5.0/topics/db/models/>
* **Tipos de campos de modelo (Field types):** Aquí encontrarás detalles sobre CharField, TextField, BooleanField, IntegerField, ImageField, SlugField, DateTimeField, y más.<https://docs.djangoproject.com/en/5.0/ref/models/fields/>
* **Relaciones de modelos (Foreign Key):** Esencial para entender cómo se conectan tus modelos Articulo, Categoria y User. [https://docs.djangoproject.com/en/5.0/topics/db/models/#many-to-one-relationships](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/topics/db/models/%23many-to-one-relationships)

### **8. Django Admin Site.** <https://docs.djangoproject.com/en/5.0/ref/contrib/admin/>

**Opciones de ModelAdmin,** personalizaciones.<https://docs.djangoproject.com/en/5.0/ref/contrib/admin/#modeladmin-options>

### **9. Categorias como app, categorias compartidas con tipos**

def \_\_str\_\_(self):

# Para una mejor representación, puedes listar los tipos asociados

tipos\_str = ", ".join([t.nombre for t in self.tipos.all()])

return f"{self.nombre} ({tipos\_str})" if tipos\_str else self.nombre

Este método \_\_str\_\_ es una función especial en Python que define cómo se representa un **objeto** de tu modelo Categoria como una **cadena de texto (string)**. Django lo usa en varios lugares, siendo el más común el **panel de administración de Django** y cuando imprimes un objeto en la consola.

**¿Para qué sirve?**

Imagina que tenés una categoría llamada "Música".

* **Sin este \_\_str\_\_ personalizado:** Si solo tuvieras return self.nombre, en el admin verías "Música". Pero no sabrías si esa "Música" es para artículos, eventos, o ambos.
* **Con este \_\_str\_\_ personalizado:**
  1. self.tipos.all(): Accede a todos los objetos TipoCategoria (ej. "Artículo", "Evento") que están relacionados con esta categoría "Música" a través de la relación Many-to-Many (tipos).
  2. [t.nombre for t in self.tipos.all()]: Esto crea una lista de los nombres de esos tipos. Si "Música" está asociada a "Artículo" y "Evento", esta lista sería ['Artículo', 'Evento'].
  3. ", ".join(...): Une los elementos de esa lista con una coma y un espacio. Entonces, ['Artículo', 'Evento'] se convierte en "Artículo, Evento". Esto se guarda en la variable tipos\_str.
  4. return f"{self.nombre} ({tipos\_str})" if tipos\_str else self.nombre:
     + Si tipos\_str no está vacío (es decir, la categoría tiene tipos asociados), devuelve una cadena formateada como **"NombreDeCategoria (Tipo1, Tipo2)"**. Por ejemplo: **"Música (Artículo, Evento)"**.
     + Si tipos\_str está vacío (la categoría no tiene ningún tipo asociado), solo devuelve el nombre de la categoría: **"Música"**.

**En resumen:** Este \_\_str\_\_ mejorado te permite ver de un vistazo, especialmente en el panel de administración, a qué **tipos** pertenece cada categoría, haciendo la gestión de tus categorías mucho más clara cuando tienes una relación Many-to-Many para los tipos.

—----------

### **10. Categorias filtradas**

Filtro usando la misma view que contiene todos los articulos; cultura/

Método get\_absolute\_url en el Modelo Categoria

def get\_absolute\_url(self):

# Esta URL aún apunta a la lista de artículos, pero el filtrado se hará diferente

return reverse('articulos:path\_lista\_articulos\_categoria', kwargs={'categoria\_slug': self.slug})

El método get\_absolute\_url es una convención de Django. Si un modelo tiene este método, Django puede usarlo para obtener la URL de un objeto específico.

**¿Qué hace aquí?**

1. reverse('articulos:path\_lista\_articulos\_categoria', ...): Esta función de Django genera una URL. Le estás diciendo: "Busca la URL llamada path\_lista\_articulos\_categoria dentro del espacio de nombres articulos."
2. kwargs={'categoria\_slug': self.slug}: Le pasas los argumentos necesarios para construir esa URL. En este caso, la URL path\_lista\_articulos\_categoria espera un slug de categoría, y le estás dando el slug de la categoría actual (self.slug).

**En resumen:** Este método proporciona una forma estándar y programática de obtener la URL para la página donde se listan los artículos (o elementos) que pertenecen a esta categoría específica.

—--------

Filtrado en \_\_init\_\_ del Formulario FormularioCrearArticulo

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

# ¡FILTRADO AQUÍ!

# Ahora, buscamos categorías que \*tengan\* al menos un tipo cuyo nombre sea 'Artículo'

# Esto es más flexible: si una categoría tiene 'Artículo' y 'Evento', igual aparecerá.

self.fields['categoria'].queryset = Categoria.objects.filter(tipos\_\_nombre='Artículo')

Este es el corazón de cómo logramos que, al crear o editar un artículo, el desplegable de categorías (el campo categoria en tu formulario) solo muestre las categorías que son relevantes para artículos.

**Contexto:**

* **FormularioCrearArticulo**: Este es tu formulario para crear/editar artículos.
* **self.fields['categoria']**: Accede al campo categoria de tu formulario. Como categoria es un ForeignKey a Categoria, Django por defecto creará un desplegable (<select>) que lista *todas* las categorías existentes en la base de datos.
* **.queryset**: Este atributo de un campo de formulario relacionado con un modelo (ModelChoiceField en este caso) controla qué objetos se muestran en el desplegable. Por defecto, es Categoria.objects.all().

**¿Qué hace la línea?**

self.fields['categoria'].queryset = Categoria.objects.filter(tipos\_\_nombre='Artículo')

1. Categoria.objects: Esto es el gestor de objetos para tu modelo Categoria, que te permite hacer consultas a la base de datos.
2. .filter(...): Le dice a Django que quieres seleccionar solo un subconjunto de categorías.
3. tipos\_\_nombre='Artículo': Esta es la parte crucial y se llama una **"lookup de relación"** o **"doble guion bajo (\_\_)"**.
   * tipos: Hace referencia al ManyToManyField llamado tipos en tu modelo Categoria.
   * \_\_nombre: Luego de \_\_, le estás diciendo a Django que acceda a un campo (nombre) del modelo relacionado (TipoCategoria).

**En resumen:**

Filtra todas las Categorias que tengan el **tipo con el nombre 'Artículo'**."

Esto permite tener una sola categoría "Música" que tenga los TipoCategoria "Artículo" y "Evento":

* En el el formulario de crear Articulo, la consulta Categoria.objects.filter(tipos\_\_nombre='Artículo') **incluirá "Música"**, porque "Música" está asociada con el TipoCategoria "Artículo".
* En el formulario para crear Evento, Categoria.objects.filter(tipos\_\_nombre='Evento'), esa misma categoría "Música" **también aparecere** porque está asociada con el TipoCategoria "Evento".

### **11 Modelo Categoria y TipoCategoria**

Para organizar mejor los artículos y permitir filtros más específicos, hemos implementado un sistema de **categorías** que se relaciona con **tipos de categorías**. Esto permite que una misma categoría (por ejemplo, "Música") pueda ser relevante para diferentes tipos de contenido (como "Artículos" y "Eventos").

**1. Modelo TipoCategoria**

Este modelo representa un tipo general de contenido al que una categoría puede pertenecer. Por ejemplo, "Artículo", "Evento" o "Galería".

# tu\_proyecto/apps/categorias/models.py

from django.db import models

from django.utils.text import slugify

from django.urls import reverse

class TipoCategoria(models.Model):

nombre = models.CharField(max\_length=50, unique=True, help\_text="Ej: Artículo, Evento, Galería")

slug = models.SlugField(max\_length=50, unique=True, blank=True)

class Meta:

verbose\_name = 'Tipo de Categoría'

verbose\_name\_plural = 'Tipos de Categoría'

ordering = ['nombre']

def save(self, \*args, \*\*kwargs):

if not self.slug:

self.slug = slugify(self.nombre)

super().save(\*args, \*\*kwargs)

def \_\_str\_\_(self):

return self.nombre

#### **Explicación de Campos y Métodos:**

* **nombre**: CharField que almacena el nombre del tipo (ej., "Artículo"). Es unique=True para asegurar que no haya tipos duplicados.
* **slug**: SlugField para una URL amigable. Se genera automáticamente a partir del nombre si no se provee, usando slugify.
* **Meta Class**: Define metadatos para el modelo:
  + verbose\_name: Nombre singular legible para humanos.
  + verbose\_name\_plural: Nombre plural legible para humanos.
  + ordering: Ordena los objetos por nombre por defecto.
* **save(self, \*args, \*\*kwargs)**: Este método se sobrescribe para generar el slug automáticamente antes de guardar el objeto, si el slug aún no existe.
* **\_\_str\_\_(self)**: Define la representación en cadena del objeto (ej., "Artículo"). Es lo que verás en el panel de administración de Django.

#### **Documentación relevante:**

* **CharField**: [Django Docs - CharField](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/ref/models/fields/%23charfield)
* **SlugField**: [Django Docs - SlugField](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/ref/models/fields/%23slugfield)
* **Model.save()**: [Django Docs - Overriding Model Methods](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/topics/db/models/%23overriding-predefined-model-methods)
* **\_\_str\_\_ method**: [Django Docs - Model.**str**()](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/ref/models/instances/%23str)
* **Meta options**: [Django Docs - Model Options](https://docs.djangoproject.com/en/5.0/ref/models/options/)

**2. Modelo Categoria**

Este modelo representa las categorías reales (ej., "Música", "Deportes", "Cocina"). Una categoría puede estar asociada a **múltiples** TipoCategoria.

# tu\_proyecto/apps/categorias/models.py

# ... (Modelo TipoCategoria arriba) ...

class Categoria(models.Model):

nombre = models.CharField(max\_length=100, unique=True)

slug = models.SlugField(max\_length=100, unique=True, blank=True)

# Relación Many-to-Many con TipoCategoria.

# Una Categoria puede tener muchos TipoCategoria, y un TipoCategoria puede tener muchas Categorias.

tipos = models.ManyToManyField(

TipoCategoria,

related\_name='categorias',

blank=True,

help\_text="Selecciona los tipos a los que pertenece esta categoría (ej. Artículo, Evento)."

)

class Meta:

verbose\_name = 'Categoría'

verbose\_name\_plural = 'Categorías'

ordering = ['nombre']

def save(self, \*args, \*\*kwargs):

if not self.slug:

self.slug = slugify(self.nombre)

super().save(\*args, \*\*kwargs)

def \_\_str\_\_(self):

# Esta representación muestra el nombre de la categoría y sus tipos asociados.

# Por ejemplo, "Música (Artículo, Evento)" o "Deportes (Artículo)".

tipos\_str = ", ".join([t.nombre for t in self.tipos.all()])

return f"{self.nombre} ({tipos\_str})" if tipos\_str else self.nombre

def get\_absolute\_url(self):

# Genera la URL para la página donde se listan los artículos de esta categoría.

# La vista 'lista\_articulos' de la app 'articulos' será la encargada de filtrar.

try:

return reverse('articulos:path\_lista\_articulos\_categoria', kwargs={'categoria\_slug': self.slug})

except:

return "/"

#### **Explicación de Campos y Métodos Adicionales:**

* **tipos**: ManyToManyField con TipoCategoria.
  + Indica que una Categoria puede estar relacionada con múltiples TipoCategoria, y viceversa.
  + **related\_name='categorias'**: Permite acceder a las categorías desde una instancia de TipoCategoria (ej., tipo\_articulo.categorias.all()).
  + **blank=True**: Permite que una categoría no tenga ningún tipo asignado.
* **\_\_str\_\_(self) (Desglose Detallado)**:
  + self.tipos.all(): Obtiene un QuerySet de todos los objetos TipoCategoria asociados a la instancia de Categoria actual.
  + [t.nombre for t in self.tipos.all()]: Esto es una **comprensión de lista**. Itera sobre cada TipoCategoria (t) en el QuerySet y crea una nueva lista que solo contiene el atributo nombre de cada tipo (ej., ['Artículo', 'Evento']).
  + ", ".join(...): Este es un método de cadena que une todos los elementos de una secuencia (la lista de nombres de tipos) en una sola cadena, usando ", " como separador. Si la lista es ['Artículo', 'Evento'], el resultado es "Artículo, Evento".
  + return f"{self.nombre} ({tipos\_str})" if tipos\_str else self.nombre: Formatea la cadena final. Si tipos\_str tiene contenido (es decir, la categoría tiene tipos asociados), devuelve "NombreCategoria (Tipo1, Tipo2)". Si no tiene tipos, devuelve solo "NombreCategoria".
* **get\_absolute\_url(self)**:
  + Una convención de Django para obtener la URL canónica de un objeto.
  + reverse('articulos:path\_lista\_articulos\_categoria', kwargs={'categoria\_slug': self.slug}): Construye la URL utilizando el nombre de la URL definida en articulos/urls.py y pasando el slug de la categoría actual como argumento.
  + El bloque try-except es una precaución para manejar casos donde la URL path\_lista\_articulos\_categoria podría no estar definida todavía, devolviendo una URL por defecto (/).

#### **Documentación relevante:**

* **ManyToManyField**: [Django Docs - ManyToManyField](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/ref/models/fields/%23manytomanyfield)
* **Lookups que cruzan relaciones (\_\_)**: [Django Docs - Lookups that cross relationships](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/ref/models/querysets/%23lookups-that-span-relationships)
* **List Comprehensions (Python)**: [Python Docs - List Comprehensions](https://www.google.com/search?q=https://docs.python.org/3/tutorial/datastructures.html%23list-comprehensions)
* **String join() method (Python)**: [Python Docs - str.join()](https://www.google.com/search?q=https://docs.python.org/3/library/stdtypes.html%23str.join)
* **reverse() function**: [Django Docs - reverse()](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/ref/urlresolvers/%23django.urls.reverse)
* **get\_absolute\_url() method**: [Django Docs - get\_absolute\_url()](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.0/ref/models/instances/%23get-absolute-url)

### 

### **12. Cambios en views articulo, para filtros**

def lista\_articulos(request, categoria\_slug=None): #puede tener o no una categoria articulos/cat

todos\_los\_articulos = Articulo.objects.all().order\_by('-creado') # \*\*\* nuevos primero

categoria\_actual = None

if categoria\_slug: #si categoriaslug tiene un valor,

categoria\_actual = get\_object\_or\_404(Categoria, slug=categoria\_slug) #busca cat o da error404

todos\_los\_articulos = todos\_los\_articulos.filter(categoria=categoria\_actual) #filtra todos los art con current cat

articulos\_por\_pagina = 12

paginator = Paginator(todos\_los\_articulos,articulos\_por\_pagina) #le doy todos los art, luego cuantos quiero por pag

page\_number = request.GET.get('page') #obtiene el numero de pag de la url y si no hay es la primera

page\_obj = paginator.get\_page(page\_number) #Obtenemos el objeto Page (una parte de la lista de art para la página actual)

todas\_las\_categorias = Categoria.objects.all().order\_by('nombre')

contexto = {

'page\_obj': page\_obj,

'categoria\_actual': categoria\_actual,

'categorias': todas\_las\_categorias

}

return render(request, 'articulos/lista\_articulos.html', contexto)

-categoria\_slug=None: Este **nuevo parámetro**. Indica que la vista ahora puede recibir opcionalmente un slug de categoría.

Si la URL no incluye un slug de categoría (ej., /articulos/), categoria\_slug será None.

Si la URL incluye un slug (ej., /articulos/categoria/musica/), Django captura "musica" y lo pasará como categoria\_slug.

* categoria\_actual = None (variable)

-

if categoria\_slug:

categoria\_actual = get\_object\_or\_404(Categoria, slug=categoria\_slug) 4

todos\_los\_articulos = todos\_los\_articulos.filter(categoria=categoria\_actual)

si tiene una cat, la busca y si no la encuentra devuelve error, cuando encuentra fitlra todos por la categoria que ahora es categoria actual

todas\_las\_categorias = Categoria.objects.all().order\_by('nombre') #esto solo obtieen todas las cat y las ordena

### **13. Comentarios app - rel login required, users, articulos**

if request.method == 'POST':

form = ComentarioForm(request.POST)

if form.is\_valid():

comentario = form.save(commit=False)

comentario.articulo = articulo

comentario.autor = request.user

comentario.save()

return redirect('articulos:detalle\_articulo', slug=articulo.slug)

**comentario = form.save(commit=False)**: Este es un patrón común en Django. form.save() normalmente guarda el objeto en la base de datos. Sin commit=False, form.save() guardaría el objeto inmediatamente. Pero como necesitamos asignar el articulo y el autor (que no vienen del formulario directamente), primero creamos el objeto en memoria (commit=False), asignamos los valores adicionales, y luego lo guardamos explícitamente con comentario.save()

\*eliminar comentario (line 146 delete\_comentario.html)

comentario.contenido|linebreaksbr

linebreaksbr reemplaza los salto de lineas por tags html <br> para mantener el formato original posteado por el usuario en el textarea del form  
  
**Users Auth Requisito de logueo y identificación de grupo**

### **Para Vistas Basadas en Funciones (con Decoradores)**

* **@login\_required**: Este es el decorador que usamos para asegurar que solo los usuarios autenticados puedan acceder a una vista.
  + [https://docs.djangoproject.com/en/5.2/topics/auth/default/#the-login-required-decorator](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.2/topics/auth/default/%23the-login-required-decorator)
* **@user\_passes\_test**: Este decorador es más flexible. Te permite definir una función personalizada para chequear cualquier condición, como si el usuario pertenece a un grupo específico.
  + [https://docs.djangoproject.com/en/5.2/topics/auth/default/#django.contrib.auth.decorators.user\_passes\_test](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.2/topics/auth/default/%23django.contrib.auth.decorators.user_passes_test)

### **Para Vistas Basadas en Clases (con Mixins)**

* **LoginRequiredMixin**: Es el equivalente en forma de clase a @login\_required.
  + [https://docs.djangoproject.com/en/5.2/topics/auth/default/#django.contrib.auth.mixins.LoginRequiredMixin](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.2/topics/auth/default/%23django.contrib.auth.mixins.LoginRequiredMixin)
* **UserPassesTestMixin**: Te permite usar una función de prueba (test\_func) dentro de tu clase para validar condiciones personalizadas, como verificamos si el usuario es is\_staff.
  + [https://docs.djangoproject.com/en/5.2/topics/auth/default/#django.contrib.auth.mixins.UserPassesTestMixin](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.2/topics/auth/default/%23django.contrib.auth.mixins.UserPassesTestMixin)

**Grupos y Permisos:** [https://docs.djangoproject.com/en/5.2/topics/auth/default/#groups](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.2/topics/auth/default/%23groups)

**Permisos de Modelos:** [https://docs.djangoproject.com/en/5.2/topics/auth/default/#model-permissions](https://www.google.com/search?q=https://docs.djangoproject.com/en/5.2/topics/auth/default/%23model-permissions)

### **14. Context Processors**

Un **context processor** en Django es una función que agrega variables al contexto de **todas las plantillas** renderizadas en tu proyecto. Esto es útil para datos que necesitas en muchas páginas, como el menú de navegación con categorias\_cultura en tu plantilla base (base.html). En tu caso:

* **Problema**: El submenú de "Cultura" en base.html usa {% for cat in categorias\_cultura %}, pero categorias\_cultura no está disponible en todas las vistas (por ejemplo, en detalle\_articulo), lo que hace que el submenú no se muestre en esas páginas.
* **Solución con Context Processor**: Creas una función que ejecuta la consulta Categoria.objects.filter(tipos\_\_nombre='Cultura').order\_by('nombre') y agrega categorias\_cultura (y otras categorías) al contexto de todas las plantillas. Así, el submenú funciona en todas las páginas sin repetir código en cada vista.
* **Ventaja**: Evita agregar las mismas consultas (categorias\_cultura, etc.) en cada vista (lista\_articulos, detalle\_articulo, etc.), haciendo el código más limpio y escalable.

**Cómo Funciona**:

1. Defines una función en articulos/context\_processors.py que devuelve un diccionario con las variables que quieres (por ejemplo, categorias\_cultura).
2. La registras en settings.py bajo TEMPLATES['OPTIONS']['context\_processors'].
3. Django ejecuta esta función para cada solicitud, y las variables están disponibles en todas las plantillas.

<https://docs.djangoproject.com/en/5.1/ref/templates/api/#writing-your-own-context-processors>

<https://docs.djangoproject.com/en/5.1/ref/templates/api/#template-context>

Contacto con 2 mails pero 1 null y blank, 1 oficial 1 responsable

### **15. Contacto forms/form.model**

<https://docs.djangoproject.com/en/5.2/topics/email/>

### **forms.Form vs. forms.ModelForm**

La diferencia principal está en cómo Django maneja los datos.

1. **forms.ModelForm**:
   * **Uso:** Lo utilizas cuando quieres que los datos del formulario se guarden directamente en la base de datos, en un modelo específico (Articulo, Nosotros, etc.).
   * **Ventaja:** Es muy rápido y automático. Tú defines model = ... y Django se encarga de crear los campos del formulario por ti, basándose en los campos de tu modelo. Por eso, en tus otros forms.py solo tenías el class Meta.
2. **forms.Form**:
   * **Uso:** Lo utilizas cuando los datos del formulario **no van a ir a la base de datos**, sino que se usarán para otra cosa, como enviar un email.
   * **Ventaja:** Te da un control total sobre cada campo, sin estar limitado por la estructura de un modelo. Tienes que definir cada campo de forma manual (nombre = forms.CharField(...), etc.).

widget - attrs (atributos)

widget=forms.Textarea(attrs={'placeholder': 'Tu mensaje'})

Django traduce eso a la siguiente etiqueta HTML: <textarea name="mensaje" id="id\_mensaje" placeholder="Tu mensaje"></textarea>

**Config en settings/local** (no esta hecho pero es lo quedeberia agregar para que funcione)

EMAIL\_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'

EMAIL\_HOST = 'smtp.gmail.com' # O el servidor SMTP de tu proveedor

EMAIL\_PORT = 587 EMAIL\_USE\_TLS = True

EMAIL\_HOST\_USER = 'tucorreo@gmail.com' # Tu dirección de correo

EMAIL\_HOST\_PASSWORD = 'tupassworddeaplicacion' # La contraseña o token de aplicación de tu correo

DEFAULT\_FROM\_EMAIL = 'tucorreo@gmail.com' # La dirección que se usará por defecto para los envíos

#### **Explicación de los campos de settings:**

* **EMAIL\_BACKEND**: Le dice a Django que use un servidor SMTP para enviar los correos.
* **EMAIL\_HOST**: La dirección del servidor SMTP.
* **EMAIL\_PORT**: El puerto de conexión. 587 es el puerto estándar para TLS.
* **EMAIL\_USE\_TLS**: Habilita la seguridad de la conexión. Es crucial para una conexión segura.
* **EMAIL\_HOST\_USER**: El email desde el que se enviarán los correos.
* **EMAIL\_HOST\_PASSWORD**: La contraseña para este email. **¡Muy importante!** Si usas Gmail, no es la contraseña de tu cuenta, sino una [contraseña de aplicación](https://support.google.com/accounts/answer/185833?hl=es) que debes generar.

Filtro eventos pasados - Python  
  
https://docs.python.org/3.13/library/datetime.html