

Django Model Utils

Casi siempre se va a necesitar en los modelos de los atributos fecha de creación y fecha de modificación.

Sin embargo es algo que se debe hacer en todos los modelos y para no estar repitiendo código pues nos apoyaremos de la herramienta Django Model Utils.

pip install django-model-utils

al instalar esta app no es necesario añadirla dentro de los entornos ya que es una aplicación que ya la trae en realidad Django sólo que se necesitamos activarlo.

Entonces al instalarlo automáticamente ya se lo ha hecho dentro de todo lo que es Django, por ende para usarlo solo se lo debe importar

get_context_data

Como su nombre indica obtiene los datos del contexto. Son los datos, variables, objetos, etc, que le vas a pasar al template para maquetarlos. Por ejemplo es donde pasarías un formulario en caso de que necesites un template con 2 o más formularios.

A menudo, es necesario presentar información adicional además de la proporcionada por la vista genérica. Por ejemplo, piense en mostrar una lista de todos los libros en cada página de detalles del editor.

La `DetailView` vista genérica proporciona al editor el contexto, pero ¿cómo obtenemos información adicional en esa plantilla?

La respuesta es subclassificar `DetailView` y proporcionar su propia implementación del `get_context_data` método. La implementación predeterminada agrega el objeto que se muestra a la plantilla, pero puede anularlo para enviar más:

```
from django.views.generic import DetailView
from books.models import Book, Publisher

class PublisherDetailView(DetailView):

    model = Publisher

    def get_context_data(self, **kwargs):
        # Call the base implementation first to get a context
        context = super().get_context_data(**kwargs)
        # Add in a QuerySet of all the books
        context['book_list'] = Book.objects.all()
        return context
```

este es el nombre de mi contexto
y con este le llamo en el html

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

    def get_context_data(self, **kwargs):
        context = super(HomePageView, self).get_context_data(**kwargs)
        # contexto de portada
        context["portada"] = Entry.objects.entrada_en_portada() #Contexto/Variable portada
        return context
```

```
<div class="cell small-12 large-6">
    <div class="card" style="width: 100%;">
        
        <div class="card-section">
            <h4>{{ portada.title }}</h4>
            <p>{{ portada.resume }}</p>
        </div>
    </div>
</div>
```

Vista para Guardar una Suscripción

Para poder hacer que cargue la url dentro del modal sin hacer uso de otro template pues lo que se hace es de la misma manera usar el POST, token y el boton submit pero además se debe poner un action en el formulario

Dentro del mismo lo que se va a poner es la url para redireccionar

```
<div class="reveal" id="exampleModal2" data-reveal style="width: 300px;">
    <p class="lead">Ingresa tu correo</p>
    <form class="grid-x grid-margin-x" method="POST" action="{% url 'home_app:add-subscription' %}"> {% csrf_token %}
        <div class="cell small-12">
            <!-- <input type="text" placeholder="E-mail.." -->
            {{form.email}}
        </div>
        <div class="cell small-12">
            <button type="submit" class="success button">Suscribirme</button>
        </div>
    </form>
</div>
```

Formulario de Contacto

En esta clase lo que se plantea es la funcionalidad del formulario en el footer que se muestra en todas las páginas, inicialmente se pensaría hacer como lo que se hizo con el Modal sin embargo se debería mandar el get context en cada una de las vistas por lo que se repetiría el código...

Cuando hacemos uso del context y en el HTML mandamos el `{{form}}` Django propiamente lo que hace es ya hacer la relación con el modelo

Sin embargo al no poder mandar el contexto del formulario `{{form.atributo}}` en el HTML debido a la redundancia de código pues lo que se hace es

maquetar manualmente los input con el id y el name para que Django los relacione con el modelo en el form

De esta forma se ahorra código y se lo ve más elegante

Context Procesor en Django

Se plantea el poder poner al lado del formulario de contacto en el footer el teléfono y correo de la página principal

Sin embargo de igual forma que en la clase anterior no debemos hacer que se repita código a pesar de que siempre va el footer en todas las páginas.

Entonces, ¿cómo hago para enviar todas las variables del teléfono y correo a todos los templates para que lo muestre?

Usaremos el Context Procesor

Primero entendamos que Django al ejecutarse con sus entornos lo primero que ejecuta son los Middleware, luego de eso se ejecutan los processors o context processors que están en el arreglo de Templates...

Aquí vamos a intervenir para que se ejecute el código en específico que queremos que se ejecute

Entonces, dentro de la carpeta applications, crearemos un processors.py, para luego en el base.py añadirlo

De esta forma se escribe una función que vaya a ser un procesor => `def home_contact(request):`
y siempre retorna un diccionario { }

Una vez definida la función y añadida en el base.py pues para llamarla en un html lo único que se hace es llamar por la clave del diccionario dependiendo el valor que quiera, por ejemplo:
`{{ correo }}`

Pantalla para el detalle de una entrada

Para que la información de mi modelo entrada, es decir el contenido se muestre correctamente y no con etiquetas html pues se hace uso de un template tag `>= tag | safe`

Funcionalidad Registro de Usuario

Al momento de trabajar con usuarios y en formularios con contraseña es importante indicar que la info esté completamente resguardada, para ello ponemos el `enctype="multipart/form-data"`

En el formulario para que los input que son de fecha salgan como tipo fecha pues se configura en los widgets

```
'date_birth': forms.DateInput(
    attrs= {
        'type': 'date'
    }
),
```

Vistas para blog favoritos de usuario

Lo que se quiere hacer es poder añadir una entrada a favoritos y para ello tengo un botón. Se pensaría inicialmente usar el Create View, sin embargo no tengo un `<form> </form>` entonces no se puede usar ni tampoco el FormView porque necesito un formulario.

¿Cómo solucionarlo?

En sí solo se necesita ejecutar el método POST al momento de dar click en el botón, se tiene también el usuario de la sesión y también el ID de la entrada. Entonces utilizaremos simplemente la vista padre View

Ojo que de todas maneras ya luego se puso la etiqueta form en el html para poder hacer el método POST

Generar un SlugField

En el modelo para la entrada habíamos añadido un atributo slug el cual lo pusimos de tipo `model.SlugField`, el cual nos sirve para trabajar con lo que sería el SEO y el posicionamiento, con las URLs generadas automáticamente.

Django ya reconoce el Slug al momento de ponerlo en el modelo, pero también nos da la libertad de que nosotros definamos como queremos que se genere este slug

Ya sea que por ejemplo queramos que la url inicie con una palabra específica, pues se tiene esa libertad para generar las URLs

Para generarlo es muy sencillo, sobreescribimos la función save en el modelo

Una característica muy importante de el slugfield es que debe ser UNICO, no se lo pone en el atributo , django ya lo asume

Sin embargo para que no exista un conflicto con las url de otro modelo que también tenga un SLUG lo que se hará es lo siguiente

- Usar el Tiempo (from datetime import timedelta, datetime) para generar un identificador único para cada registro un número en base al tiempo actual
- Sumado a eso apoyandonos de la librería slugify pues generaremos la url en base a un texto concatenado, en éste caso en base al título (atributo de mi modelo)

Slug como parámetro de la URL

Al momento de mandar parámetros por ejemplo en el DetailView el <pk> pues también se puede mandar como parámetro el <slug> recuperando de igual forma el objeto

Si se manda el slug como parámetro pues el mismo busca dentro de nuestros modelos un slugfile y recupera el registro en base a ese slugfile, es por esta razón que siempre debe ser único

Ya en el html simplemente se lo llama y funcionará como si fuera el pk

Block Title para el SEO

Se modificó el html para que el título de la página lo obtenga desde el modelo, de esta forma se mejora el posicionamiento en el SEO

Sumado a esto también se modificó el base.html para poder también mostrar la descripción de la página, para ello se añadió el bloque de description y dentro del mismo se puso un meta en el cual en su parámetro name se pone el nombre y en el contenido iría el atributo description del modelo.

Teoría Sitemap SEO Django

¿Cómo hago para hacerle a entender al motor de búsqueda cómo está estructurado mi sitio web?

Si le brindamos toda esa información ya bien estructurada al motor de búsqueda, va a ser mucho más sencillo que el motor de búsqueda entienda que es X página web donde está lo que el usuario está buscando

Necesitamos darle ésa facilidad al motor de búsqueda ==> a éso le llamamos un Sitemap

Implementando Sitemap

Django ya trae herramientas para los Sitemap, creamos los sitemaps.py y aquí iremos creando las diferentes clases dependiendo de los modelos con los cuales queramos implementar el Sitemap.

* Recordar que Django pone como requisito el sobrescribir el Sitemap para poder implementar

- Ya en las urls.py dividiríamos la estructura , crearemos el objeto sitemap que genera xml a fin de poder indicar que se genere la estructura en base a nuestro sitemap creado especificando cual va a ser la pag principal y cuales van a ser las secundarias
- Sumado a éso también indicaríamos las URLS generadas (indicando el nombre de la página, que es una estructura sitemap , y la estructura sitemap nuestra
- Finalmente concatenamos todas las URLS

Completando el código para los Sitemap

Al momento de hacer los sitemap y trabajar con los items para que ésto funcione correctamente se necesita sobrescribir una función más dentro del modelo

la cual se llama

`get_absolute_url`

```
def get_absolute_url(self):
    """ # Aquí tenemos que definir cómo es que se crea la url para cada una de las entradas, entonces revisando mi urls.py
        # se tiene ...
        # 'entrada/<slug>', views.EntryDetailView.as_view(), name='entry-detail'
        # Entonces esta es la URL que necesitaremos para el posicionamiento

    return reverse_lazy(
        'entrada_app:entry-detail',
        kwargs = { # con el kwargs mando el parametro de la url que en este caso seria el slug
            'slug': self.slug
        }
    )
```

Si bien cada Sitemap está asociado a un modelo, no siempre se requerirá que un sitemap esté asociado a un modelo. por ende

¿Qué pasa con los que no dependen exclusivamente de un modelo? es decir los que son sólo páginas planas que nosotros queremos jalar la URL, como en el caso del index?

Necesitamos indicarle un periodo de cambio o un periodo de fecha de creación.


Para ello en nuestro archivo de sitemaps.py debemos agregar algunas funciones más

- lastmod <== lo usamos para indicar un periodo de fecha de creación , por defecto es el día
- location <== # Hace referencia a como genera O donde está la URL

A parte de éso también en mi entorno debo importar los sitemaps, sumado a éso también en el HTML también debo indicar el title y la etiqueta meta.

Sin embargo no vamos a necesitar solo esto para el posicionamiento sino que también vamos a necesitar indicar que del HTML a qué palabras clave está relacionado independientemente de si se encuentren en el artículo o no...

Esto se lo conoce como TAGS



Atributo que se añadió al modelo Entrada con relacion Many to Many

Entonces ésto simplemente lo añadimos al html puesto que ya es un atributo del modelo, pero para añadirlo necesitaríamos otro bloque en el base.html donde especificaremos los Tags

Una vez puesto eso en el base.html simplemente reemplazo el block con el modelo que corresponda



