Tutorial: Construyendo un Sistema de Importación con Django

¡Bienvenido! Este documento es una guía paso a paso para construir desde cero un Sistema de Importación de Productos utilizando el framework Django. Es ideal para estudiantes que deseen aprender Django de una forma práctica y detallada, cubriendo desde la configuración inicial hasta la creación de un sistema CRUD completo.

Introducción

En este tutorial, construiremos una aplicación web que permite gestionar productos, categorías y embarques de importación. Los usuarios podrán ver, crear, editar y eliminar tanto productos como embarques a través de una interfaz web amigable construida con Bootstrap 5.

Requisitos Previos:

- Tener Python instalado en tu sistema.
- Conocimientos básicos de la línea de comandos.

Paso 1: Configuración del Entorno

Siempre es una buena práctica aislar las dependencias de un proyecto en un **entorno virtual**.

1. **Crea el entorno virtual:**

   ```
   python -m venv venv
   ```

   - **Explicación:** El comando `python -m venv venv` utiliza el módulo `venv` de Python para crear un nuevo entorno virtual. El primer `venv` es el nombre del módulo de Python, y el segundo `venv` es el nombre del directorio donde se creará el entorno virtual. Esto aísla las librerías de tu proyecto del sistema global de Python, evitando conflictos de versiones.
2. **Activa el entorno virtual:** Una vez creado, debes activar el entorno para que las instalaciones de `pip` se realicen dentro de este.

   - En Windows:

     ```
     .\venv\Scripts\activate
     ```

     - **Explicación:** Este comando ejecuta el script `activate` que se encuentra en la subcarpeta `Scripts` dentro de tu entorno virtual `venv`. Al activarlo, tu terminal utilizará el `python` y `pip` específicos de este entorno.
   - En macOS/Linux:

```
source venv/bin/activate
```

- **Explicación:** En sistemas Unix-like, `source` ejecuta el script `activate` que ajusta las variables de entorno de tu shell para apuntar al entorno virtual.

3. **Instala las dependencias:** Necesitamos Django para el framework, `django-grappelli` para una mejor interfaz de admin y `Pillow` para el manejo de imágenes.

```
pip install Django django-grappelli pillow
```

- **Explicación:** `pip` es el gestor de paquetes de Python. Con este comando, instalamos tres librerías esenciales:
  - `Django`: El framework web principal.
  - `django-grappelli`: Una aplicación que mejora la apariencia y usabilidad del panel de administración de Django.
  - `Pillow`: Una librería de procesamiento de imágenes necesaria para que `ImageField` de Django funcione correctamente.

Paso 2: Creación del Proyecto y la Aplicación

1. **Crea el proyecto Django:** Este comando crea la estructura base del proyecto en el directorio actual.

```
django-admin startproject sistema_importacion .
```

- **Explicación:** `django-admin` es la utilidad de línea de comandos de Django. `startproject` es un subcomando que genera un esqueleto de proyecto. `sistema_importacion` es el nombre de nuestro proyecto. El `.` al final indica que el proyecto debe crearse en el directorio actual, sin crear una carpeta anidada con el mismo nombre.

2. **Crea la aplicación `productos`:** En Django, un proyecto se compone de una o más "apps". Una app es un módulo que hace algo específico. Nuestra app principal se llamará `productos`.

```
python manage.py startapp productos
```

- **Explicación:** `manage.py` es una utilidad que se crea con tu proyecto Django y te permite interactuar con él. `startapp` es un subcomando para generar la estructura básica de una nueva aplicación llamada `productos`. Esta aplicación contendrá la lógica de negocio para nuestros productos, categorías y embarques.

Paso 3: Configuración del Proyecto (`sistema_importacion/settings.py`)

Ahora, debemos registrar nuestra nueva app y configurar los directorios para archivos multimedia.

Abre `sistema_importacion/settings.py` y modifica lo siguiente:

1. **Añade `grappelli` y `productos` a `INSTALLED_APPS`**. Es importante que `grappelli` vaya antes de `django.contrib.admin` para que sus estilos y funcionalidades se apliquen correctamente sobre el admin por defecto de Django.

```
# sistema_importacion/settings.py

INSTALLED_APPS = [
    'grappelli', # Debe ir antes de 'django.contrib.admin'
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'productos', # Nuestra aplicación personalizada
]
```

    - **Explicación:** La lista `INSTALLED_APPS` le dice a Django qué aplicaciones están activas en tu proyecto. Al añadir `grappelli` y `productos`, Django sabe que debe cargar sus modelos, vistas, plantillas, etc. La posición de `grappelli` es crucial para que sobrescriba los estilos del administrador.

2. **Añade la configuración para `MEDIA_URL` y `MEDIA_ROOT`** al final del archivo. Esto le dice a Django dónde se guardarán físicamente los archivos subidos por el usuario (como las imágenes de los productos) y cómo se accederá a ellos a través de URLs.

```
# sistema_importacion/settings.py

import os # Asegúrate de que esta línea esté al inicio del archivo si no lo está

# ... (código existente) ...

# Media files (archivos subidos por los usuarios)
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
```

    - **Explicación:**
        - `MEDIA_URL`: Es la URL base que servirá los archivos multimedia. Por ejemplo, una imagen subida podría estar disponible en `http://localhost:8000/media/productos/mi_imagen.jpg`.
        - `MEDIA_ROOT`: Es la ruta absoluta en el sistema de archivos donde Django buscará estos archivos multimedia. `os.path.join(BASE_DIR, 'media')` crea una carpeta llamada `media` en la raíz de tu proyecto.

Paso 4: Creación de los Modelos (`productos/models.py`)

Los modelos son la representación de las tablas de nuestra base de datos en código Python. Definen la estructura de los datos que nuestra aplicación almacenará.

Abre `productos/models.py` y escribe el siguiente código:

```
# productos/models.py
```

```
from django.db import models

class Categoria(models.Model):
    # Campo para el nombre de la categoría, con un máximo de 100 caracteres
    nombre = models.CharField(max_length=100)
    # Campo para una descripción más larga, opcional (puede estar en blanco o ser nulo en la BD)
    descripcion = models.TextField(blank=True, null=True)

    def __str__(self):
        # Representación legible del objeto Categoria
        return self.nombre

class Producto(models.Model):
    # Nombre del producto
    nombre = models.CharField(max_length=200)
    # Precio del producto, con 10 dígitos en total y 2 decimales
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    # Cantidad de productos en stock
    stock = models.IntegerField()
    # Campo para subir una imagen. Las imágenes se guardarán en 'media/productos/'. Opcional.
    imagen = models.ImageField(upload_to='productos/', blank=True, null=True)
    # Clave foránea a Categoria. Cuando una Categoría se elimina, sus Productos también (CASCADE
    categoria = models.ForeignKey(Categoria, on_delete=models.CASCADE)

    def __str__(self):
        # Representación legible del objeto Producto
        return self.nombre

class Embarque(models.Model):
    # Código de rastreo único para cada embarque
    codigo_rastreo = models.CharField(max_length=100, unique=True)
    # Fecha estimada de llegada del embarque
    fecha_llegada = models.DateField()
    # Relación muchos a muchos con Producto. Un embarque tiene varios productos y un producto pu
    # 'related_name' permite acceder a los embarques desde un producto (ej: producto.embarques.a
    productos = models.ManyToManyField(Producto, related_name='embarques')

    def __str__(self):
        # Representación legible del objeto Embarque
        return self.codigo_rastreo
```

- **Explicación de los campos:**
  - `models.CharField`: Para textos cortos (requiere `max_length`).
  - `models.TextField`: Para textos largos, como descripciones.
  - `models.DecimalField`: Para números decimales, ideal para monedas (requiere `max_digits` y `decimal_places`).
  - `models.IntegerField`: Para números enteros.
  - `models.ImageField`: Para subir imágenes (requiere `Pillow` y `upload_to` para especificar la subcarpeta dentro de `MEDIA_ROOT`).
  - `models.DateField`: Para fechas.
  - `models.ForeignKey`: Establece una relación de uno a muchos. `on_delete=models.CASCADE` significa que si la categoría a la que pertenece un producto se elimina, el producto también se elimina.
  - `models.ManyToManyField`: Establece una relación de muchos a muchos. `related_name='embarques'` es útil para consultar los embarques asociados a un producto.
- **`blank=True, null=True`:** Hace que un campo sea opcional en el formulario (`blank=True`) y permite que sea nulo en la base de datos (`null=True`).

- **`unique=True`:** Asegura que no haya dos embarques con el mismo `codigo_rastreo`.
- **`__str__(self)`:** Un método especial que define cómo se representa un objeto del modelo como una cadena de texto. Es muy útil en el panel de administración.

Paso 5: Migraciones de la Base de Datos

Una vez que hemos definido o modificado nuestros modelos, necesitamos informarle a la base de datos sobre estos cambios para que se creen las tablas correspondientes.

1. **Crea el archivo de migración:**

   ```
   python manage.py makemigrations productos
   ```

   - **Explicación:** Este comando le pide a Django que examine los cambios realizados en el archivo `productos/models.py`. Si encuentra diferencias entre el estado actual de los modelos y la última migración aplicada, generará un nuevo archivo en el directorio `productos/migrations/`. Este archivo contendrá el código Python necesario para crear, modificar o eliminar tablas y campos en la base de datos para que coincidan con la definición de tus modelos.
2. **Aplica la migración:**

   ```
   python manage.py migrate
   ```

   - **Explicación:** Este comando ejecuta las migraciones pendientes en tu proyecto. Es decir, toma los archivos de migración generados (incluyendo el que acabas de crear) y aplica los cambios a la base de datos subyacente. Esto creará las tablas `Categoria`, `Producto` y `Embarque` en tu archivo `db.sqlite3` (o la base de datos que estés usando).

Paso 6: El Panel de Administración de Django

Django ofrece un panel de administración potente y personalizable que se genera automáticamente a partir de tus modelos. Para que nuestros modelos aparezcan en este panel, debemos registrarlos.

1. **Configura `productos/admin.py`:** Abre el archivo `productos/admin.py` y añade el siguiente código para registrar tus modelos y personalizar la interfaz.

   ```
   # productos/admin.py

   from django.contrib import admin
   from .models import Categoria, Producto, Embarque

   class EmbarqueAdmin(admin.ModelAdmin):
       # Mejora la interfaz para seleccionar productos en un embarque, mostrando un selector
       filter_horizontal = ('productos',)

   # Registra el modelo Categoria con la interfaz de administración
   ```

```
admin.site.register(Categoria)
# Registra el modelo Producto con la interfaz de administración
admin.site.register(Producto)
# Registra el modelo Embarque usando la clase de personalización EmbarqueAdmin
admin.site.register(Embarque, EmbarqueAdmin)
```

- **Explicación:**
  - `admin.site.register()`: Esta función es fundamental para que tus modelos sean visibles y gestionables en el panel de administración.
  - `EmbarqueAdmin`: Al crear una clase que hereda de `admin.ModelAdmin`, podemos personalizar cómo se muestra un modelo en el admin. `filter_horizontal = ('productos',)` es una opción que mejora la usabilidad de los campos `ManyToManyField`, presentando una interfaz más amigable para seleccionar múltiples elementos.

2. **Crea un superusuario:** Necesitarás una cuenta de administrador para acceder y gestionar el contenido a través del panel de administración.

```
python manage.py createsuperuser
```

- **Explicación:** Este comando te guiará a través de la creación de un usuario con todos los permisos (superusuario). Se te pedirá un nombre de usuario, una dirección de correo electrónico (opcional) y una contraseña (la cual no se mostrará mientras la escribes). Este usuario será el que utilices para iniciar sesión en `http://127.0.0.1:8000/admin/`.

Paso 7: Creación de Vistas y URLs

El patrón de diseño MVT (Modelo-Vista-Plantilla) es central en Django. Hasta ahora tenemos los Modelos (datos). Ahora crearemos las Vistas (lógica de negocio) y las URLs (rutas de acceso).

1. **Configura las URLs del proyecto (`sistema_importacion/urls.py`):** Este es el archivo principal de URLs de tu proyecto. Aquí le indicamos a Django cómo enrutar las peticiones web a las diferentes aplicaciones.

```
# sistema_importacion/urls.py

from django.contrib import admin
from django.urls import path, include
from django.conf import settings
from django.conf.urls.static import static # Importa para servir archivos media

urlpatterns = [
    path('grappelli/', include('grappelli.urls')), # Rutas de grappelli
    path('admin/', admin.site.urls), # Rutas del panel de administración
    path('', include('productos.urls')), # Incluye las rutas de nuestra app 'productos'
]

# Sirve archivos media solo durante el desarrollo (DEBUG=True)
if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

- **Explicación:**

- ♠ `path('grappelli/', include('grappelli.urls'))`: Incluye las URLs definidas por `django-grappelli`.
- ♠ `path('admin/', admin.site.urls)`: Define la ruta para acceder al panel de administración de Django.
- ♠ `path('', include('productos.urls'))`: Le dice a Django que, para la ruta raíz (`''`), busque las definiciones de URL en el archivo `productos/urls.py` de nuestra aplicación.
- ♠ `if settings.DEBUG: ...`: Esta sección es crucial para el desarrollo. Permite que Django sirva los archivos subidos por el usuario (MEDIA) desde la carpeta `MEDIA_ROOT`. **¡No usar en producción!**

2. **Crea el archivo `productos/urls.py`:** Este archivo contendrá las definiciones de URL específicas para nuestra aplicación `productos`.

```
# productos/urls.py

from django.urls import path
from . import views # Importa las vistas de nuestra app

app_name = 'productos' # Define un espacio de nombres para las URLs de esta app

urlpatterns = [
    path('', views.lista_productos, name='lista_productos'), # Ruta para el listado de pr
]
```

   - **Explicación:**
     - ♠ `app_name = 'productos'`: Define un espacio de nombres para las URLs de esta aplicación. Esto permite referenciar las URLs como `productos:lista_productos`, lo que ayuda a evitar conflictos de nombres si otras apps tuvieran una URL con el mismo nombre.
     - ♠ `path('', views.lista_productos, name='lista_productos')`: Asocia la ruta raíz de nuestra aplicación (`''`) con la función `lista_productos` de `views.py`. El `name='lista_productos'` es un alias para esta URL, que es muy útil para referenciarla desde las plantillas y el código Python.

3. **Crea la vista `lista_productos` en `productos/views.py`:** Esta vista es una función sencilla que recupera datos de nuestros modelos y los pasa a una plantilla para su renderizado.

```
# productos/views.py

from django.shortcuts import render # Para renderizar plantillas
from .models import Producto # Importa el modelo Producto

def lista_productos(request):
    """
    Muestra una lista de todos los productos, optimizando la consulta
    para incluir los embarques asociados.
    """
    # Obtiene todos los productos y pre-carga sus embarques asociados para optimizar las
    productos = Producto.objects.all().prefetch_related('embarques')
    context = {
        'productos': productos # Pasa el queryset de productos a la plantilla
    }
    # Renderiza la plantilla 'productos/lista.html' con los datos del contexto
    return render(request, 'productos/lista.html', context)
```

   - **Explicación:**
     - ♠ `render(request, 'template_name', context)`: Una función de atajo para cargar

una plantilla, pasarle un contexto y devolver un objeto `HttpResponse` con la plantilla renderizada.

- `Producto.objects.all()`: Recupera todos los objetos (registros) del modelo `Producto`.
- `prefetch_related('embarques')`: Esta es una optimización crucial. Cuando Django necesita acceder a una relación de muchos a muchos (o muchos a uno inversa) para múltiples objetos, puede hacer una consulta separada para cada objeto individual, lo que resulta en muchas consultas a la base de datos ("N+1 query problem"). `prefetch_related` le indica a Django que cargue todos los objetos relacionados de una vez en una sola consulta adicional, mejorando significativamente el rendimiento.
- `context`: Un diccionario donde las claves serán los nombres de las variables disponibles en la plantilla y los valores serán los datos que queremos pasar.

Paso 8: Creación de las Plantillas (Templates)

Las plantillas HTML son la "T" en MVT. Son archivos que definen la estructura y el diseño de la interfaz de usuario. Utilizan un lenguaje de plantillas de Django para mostrar datos dinámicos.

1. **Crea la estructura de directorios:** Asegúrate de que tienes el siguiente directorio: `productos/templates/productos/`. Django buscará plantillas en `templates/` dentro de cada aplicación.
2. **Crea la plantilla base (`productos/templates/productos/base.html`):** Esta plantilla servirá como el esqueleto principal de todas nuestras páginas. Definirá la estructura HTML básica, incluirá los archivos CSS y JS de Bootstrap, y contendrá los bloques donde las plantillas hijas insertarán su contenido específico.

```
<!doctype html>
<html lang="es">
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Sistema de Importación</title>
    <!-- Bootstrap CSS desde CDN (Content Delivery Network) -->
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/css/bootstrap.min.css" r
    <style>
      body {
        font-size: .875rem; /* Ajuste del tamaño de fuente base */
      }
      .navbar {
        background-color: #343a40; /* Color de fondo para la barra de navegación */
      }
    </style>
</head>
<body>

<!-- Cabecera de la página con la barra de navegación de Bootstrap -->
<header>
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark sticky-top shadow">
      <div class="container-fluid">
        <a class="navbar-brand" href="{% url 'productos:lista_productos' %}">Sistema Impo
        <!-- Botón de hamburguesa para colapsar el menú en móviles -->
        <button class="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-ta
```

```
          <span class="navbar-toggler-icon"></span>
        </button>
        <!-- Contenido del menú -->
        <div class="collapse navbar-collapse" id="navbarNav">
          <ul class="navbar-nav ms-auto"> <!-- ms-auto empuja los elementos a la derecha ·
            <li class="nav-item">
              <a class="nav-link" href="{% url 'productos:lista_productos' %}">Productos<,
            </li>
            <li class="nav-item">
              <a class="nav-link" href="{% url 'productos:embarque_lista' %}">Embarques</a
            </li>
            <li class="nav-item">
              <a class="nav-link" href="/admin/">Admin</a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
</header>

<div class="container-fluid">
    <div class="row">
        <main class="col-md-12 ms-sm-auto col-lg-12 px-md-4">
            <!-- Sección para el título de la página y botones de acción contextuales -->
            <div class="d-flex justify-content-between flex-wrap flex-md-nowrap align-iter
                <h1 class="h2">{% block page_title %}{% endblock %}</h1>
                {% if request.resolver_match.view_name == 'productos:lista_productos' %}
                <div class="btn-toolbar mb-2 mb-md-0">
                    <a href="{% url 'productos:producto_crear' %}" class="btn btn-sm btn-c
                        Añadir Nuevo Producto
                    </a>
                </div>
                {% endif %}
                {% if request.resolver_match.view_name == 'productos:embarque_lista' %}
                <div class="btn-toolbar mb-2 mb-md-0">
                    <a href="{% url 'productos:embarque_crear' %}" class="btn btn-sm btn-c
                        Añadir Nuevo Embarque
                    </a>
                </div>
                {% endif %}
            </div>
            {% block content %}<!-- Este bloque será rellenado por las plantillas hijas -·
        </main>
    </div>
</div>

<!-- Bootstrap Bundle con Popper JS -->
<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.0.2/dist/js/bootstrap.bundle.min.js'
</body>
</html>
```

- **Explicación:**
  - `<!doctype html>`: Define el tipo de documento HTML5.
  - `<meta charset="utf-8">`, `<meta name="viewport">`: Configuración estándar para el charset y la visualización responsiva.
  - `<title>Sistema de Importación</title>`: Título de la página que aparece en la pestaña del navegador.
  - `link` y `script` de Bootstrap: Incluyen los estilos CSS y las funcionalidades JavaScript de Bootstrap 5 desde un CDN. `integrity` y `crossorigin` son para seguridad.
  - `<header>` y `<nav>`: Contienen la barra de navegación. `navbar-expand-lg` hace que se expanda en pantallas grandes y se colapse en pantallas medianas/pequeñas. `navbar-`

`toggler` es el botón de hamburguesa. `navbarNav` es el ID del contenido colapsable.

- `{% url 'productos:lista_productos' %}`: Una etiqueta de plantilla de Django que genera la URL para la vista llamada `lista_productos` en el espacio de nombres `productos`. Es preferible usarla en lugar de URLs codificadas (`/productos/`) para mayor flexibilidad.
- `{% block page_title %}{% endblock %}`: Define un bloque que las plantillas hijas pueden sobrescribir para establecer el título de la página (`<h1>`).
- `{% block content %}{% endblock %}`: Este es el bloque principal donde las plantillas que heredan de `base.html` insertarán su contenido específico.
- `{% if request.resolver_match.view_name == 'productos:lista_productos' %}`: Lógica condicional para mostrar botones contextuales (ej. "Añadir Nuevo Producto") solo cuando se está en la vista de listado de productos.

3. **Crea la plantilla de lista (`productos/templates/productos/lista.html`):** Esta plantilla hereda de `base.html` y muestra la lista de productos en un formato de tarjetas (cards) de Bootstrap.

```
{% extends 'productos/base.html' %} {# Indica que esta plantilla hereda de base.html #}

{% block page_title %}Listado de Productos{% endblock %} {# Define el título específico pa

{% block content %} {# Inicia el bloque de contenido principal #}
<div class="row"> {# Fila de Bootstrap para organizar las tarjetas #}
    {% for producto in productos %} {# Itera sobre la lista de productos pasada desde la v
    <div class="col-md-4 col-lg-3 mb-4"> {# Columna de Bootstrap para cada tarjeta, respon
        <div class="card h-100"> {# Tarjeta de Bootstrap con altura fija #}
            {% if producto.imagen %} {# Muestra la imagen si existe #}
                <img src="{{ producto.imagen.url }}" alt="{{ producto.nombre }}" class="ca
            {% else %} {# Si no hay imagen, muestra un placeholder #}
                <div class="d-flex align-items-center justify-content-center bg-light" sty
                    <span class="text-muted">Sin imagen</span>
                </div>
            {% endif %}
            <div class="card-body"> {# Contenido de la tarjeta #}
                <h5 class="card-title">{{ producto.nombre }}</h5> {# Nombre del producto #
                <h6 class="card-subtitle mb-2 text-muted">{{ producto.categoria.nombre }}<
                <p class="card-text">
                    <strong>Embarques:</strong>
                    {% for embarque in producto.embarques.all %} {# Itera sobre los embarc
                        <span class="badge bg-secondary">{{ embarque.codigo_rastreo }}</sp
                    {% empty %} {# Si no hay embarques, muestra un mensaje #}
                        <small class="text-muted">Sin embarques asociados.</small>
                    {% endfor %}
                </p>
            </div>
            <div class="card-footer"> {# Pie de la tarjeta con botones de acción #}
                <div class="btn-group btn-group-sm" role="group">
                    <a href="{% url 'productos:producto_detalle' producto.pk %}" class="bt
                    <a href="{% url 'productos:producto_editar' producto.pk %}" class="btn
                    <a href="{% url 'productos:producto_eliminar' producto.pk %}" class="b
                </div>
            </div>
        </div>
    </div>
    {% endfor %}
</div>
{% endblock %} {# Finaliza el bloque de contenido #}
```

- o

Explicación:

- ♠ `{% extends 'productos/base.html' %}`: Esta es la clave para la herencia de plantillas. Le dice a Django que esta plantilla debe usar `base.html` como su "padre".
- ♠ `{% block page_title %}` y `{% block content %}`: Estas etiquetas sobrescriben los bloques del mismo nombre definidos en `base.html`.
- ♠ `{% for ... in ... %}`: Un bucle para iterar sobre la lista de productos.
- ♠ `{{ producto.imagen.url }}`: Accede a la URL de la imagen del producto.
- ♠ `{% if producto.imagen %}`: Una etiqueta condicional para mostrar la imagen solo si existe.
- ♠ `{{ producto.categoria.nombre }}`: Accede al nombre de la categoría a través de la relación de clave foránea.
- ♠ `{{ producto.embarques.all }}`: Accede a todos los objetos `Embarque` relacionados con el `Producto` actual, gracias al `related_name='embarques'` que definimos en el modelo.
- ♠ `{% empty %}`: Una cláusula opcional en los bucles `for` que se ejecuta si la lista sobre la que se itera está vacía.
- ♠ `{{ producto.pk }}`: Accede a la clave primaria (ID) del producto, necesaria para las URLs de detalle, edición y eliminación.

Paso 9: Implementación del CRUD (Crear, Leer, Actualizar, Borrar)

Para hacer nuestro sistema interactivo y permitir a los usuarios gestionar los productos y embarques directamente desde la interfaz, implementaremos las operaciones CRUD (Create, Read, Update, Delete). Utilizaremos los `ModelForm` de Django para los formularios y las Vistas Basadas en Clases Genéricas (CBV) para la lógica de las vistas.

1. **Crea `productos/forms.py`:** Este archivo contendrá las definiciones de nuestros formularios. Los `ModelForm` son una característica poderosa de Django que genera formularios automáticamente a partir de los modelos.

```
# productos/forms.py

from django import forms
from .models import Producto, Embarque

class ProductoForm(forms.ModelForm):
    class Meta:
        model = Producto # Asocia este formulario con el modelo Producto
        fields = ['nombre', 'precio', 'stock', 'categoria', 'imagen'] # Campos a incluir e
        widgets = { # Personaliza los widgets HTML para cada campo
            'nombre': forms.TextInput(attrs={'class': 'form-control'}),
            'precio': forms.NumberInput(attrs={'class': 'form-control'})
            'stock': forms.NumberInput(attrs={'class': 'form-control'})
            'categoria': forms.Select(attrs={'class': 'form-select'})
            'imagen': forms.ClearableFileInput(attrs={'class': 'form-control'})
        }

class EmbarqueForm(forms.ModelForm):
    class Meta:
        model = Embarque # Asocia este formulario con el modelo Embarque
        fields = ['codigo_rastreo', 'fecha_llegada', 'productos'] # Campos a incluir
```

```
        widgets = { # Personaliza los widgets HTML
            'codigo_rastreo': forms.TextInput(attrs={'class': 'form-control'})
            'fecha_llegada': forms.DateInput(attrs={'class': 'form-control', 'type': 'date
            'productos': forms.SelectMultiple(attrs={'class': 'form-select', 'size': '10'
        }
```

- **Explicación:**
  - `forms.ModelForm`: La clase base para formularios que se vinculan directamente a un modelo.
  - `class Meta`: Una clase interna que configura el `ModelForm`.
  - `model`: Especifica el modelo al que está asociado el formulario.
  - `fields`: Una tupla o lista de los campos del modelo que se incluirán en el formulario.
  - `widgets`: Permite anular los widgets HTML por defecto que Django genera para cada campo. Aquí usamos clases de Bootstrap (`form-control`, `form-select`) para estilizar los campos. Para `fecha_llegada`, `type='date'` le da al navegador un control de selección de fecha.

2. **Actualiza `productos/views.py` con Vistas Basadas en Clases (CBV):** Las CBV son clases Python que manejan peticiones web. Son muy reutilizables y reducen la cantidad de código repetitivo, especialmente para operaciones CRUD comunes.

```
# productos/views.py

from django.shortcuts import render # Para renderizar plantillas
from django.urls import reverse_lazy # Para generar URLs de forma lazy (cuando sean necesa
from django.views.generic import ListView, DetailView, CreateView, UpdateView, DeleteView
from .models import Producto, Embarque
from .forms import ProductoForm, EmbarqueForm

# ... la vista lista_productos (función) existente ...

# Vistas de Productos CRUD (Basadas en Clases)
class ProductoDetailView(DetailView):
    model = Producto # Especifica el modelo con el que trabaja esta vista
    template_name = 'productos/producto_detalle.html' # Plantilla a usar para mostrar los

class ProductoCreateView(CreateView):
    model = Producto
    form_class = ProductoForm # Especifica el formulario a usar para crear un nuevo produc
    template_name = 'productos/producto_form.html' # Plantilla que contendrá el formulario
    success_url = reverse_lazy('productos:lista_productos') # URL a la que redirigir despu

class ProductoUpdateView(UpdateView):
    model = Producto
    form_class = ProductoForm # Formulario para actualizar
    template_name = 'productos/producto_form.html' # Misma plantilla de formulario para cr
    success_url = reverse_lazy('productos:lista_productos') # Redirige al listado después

class ProductoDeleteView(DeleteView):
    model = Producto
    template_name = 'productos/producto_confirm_delete.html' # Plantilla para confirmar la
    success_url = reverse_lazy('productos:lista_productos') # Redirige al listado después

# Vistas de Embarques CRUD (Basadas en Clases)
class EmbarqueListView(ListView):
    model = Embarque
    template_name = 'productos/embarque_lista.html' # Plantilla para el listado de embarqu
    context_object_name = 'embarques' # Nombre de la variable en la plantilla que contendr
    queryset = Embarque.objects.all().prefetch_related('productos') # Consulta para obtene
```

```
class EmbarqueDetailView(DetailView):
    model = Embarque
    template_name = 'productos/embarque_detalle.html'

class EmbarqueCreateView(CreateView):
    model = Embarque
    form_class = EmbarqueForm
    template_name = 'productos/embarque_form.html'
    success_url = reverse_lazy('productos:embarque_lista')

class EmbarqueUpdateView(UpdateView):
    model = Embarque
    form_class = EmbarqueForm
    template_name = 'productos/embarque_form.html'
    success_url = reverse_lazy('productos:embarque_lista')

class EmbarqueDeleteView(DeleteView):
    model = Embarque
    template_name = 'productos/embarque_confirm_delete.html'
    success_url = reverse_lazy('productos:embarque_lista')
```

- **Explicación:**
  - `ListView`: Muestra una lista de objetos.
  - `DetailView`: Muestra los detalles de un único objeto.
  - `CreateView`: Presenta un formulario para crear un nuevo objeto y lo guarda.
  - `UpdateView`: Presenta un formulario pre-rellenado con los datos de un objeto existente para su edición.
  - `DeleteView`: Muestra una página de confirmación para eliminar un objeto.
  - `model`: La CBV usará este modelo para interactuar con la base de datos.
  - `template_name`: La plantilla que la CBV renderizará.
  - `form_class`: El formulario que usará la `CreateView` o `UpdateView`.
  - `success_url`: La URL a la que el usuario será redirigido después de una operación exitosa (crear, actualizar, eliminar). `reverse_lazy` se usa para que la URL se resuelva solo cuando sea necesario, evitando problemas de importación circular.
  - `context_object_name`: Para `ListView`, es el nombre de la variable que contendrá la lista de objetos en la plantilla (por defecto es `object_list`).

3. **Actualiza `productos/urls.py` con las nuevas rutas:** Necesitamos añadir rutas para cada una de las nuevas CBV que hemos creado.

```
# productos/urls.py

from django.urls import path
from . import views

app_name = 'productos'

urlpatterns = [
    # URLs de Productos
    path('', views.lista_productos, name='lista_productos'),
    path('producto/nuevo/', views.ProductoCreateView.as_view(), name='producto_crear'),
    path('producto/<int:pk>/', views.ProductoDetailView.as_view(), name='producto_detalle
    path('producto/<int:pk>/editar/', views.ProductoUpdateView.as_view(), name='producto_e
    path('producto/<int:pk>/eliminar/', views.ProductoDeleteView.as_view(), name='producto

    # URLs de Embarques
    path('embarques/', views.EmbarqueListView.as_view(), name='embarque_lista'),
```

```
    path('embarques/nuevo/', views.EmbarqueCreateView.as_view(), name='embarque_crear'),
    path('embarques/<int:pk>/', views.EmbarqueDetailView.as_view(), name='embarque_detalle
    path('embarques/<int:pk>/editar/', views.EmbarqueUpdateView.as_view(), name='embarque_
    path('embarques/<int:pk>/eliminar/', views.EmbarqueDeleteView.as_view(), name='embarqu
]
```

- **Explicación:**
  - `path('producto/nuevo/', views.ProductoCreateView.as_view(), name='producto_crear')`: Una URL para la creación. `as_view()` es necesario para usar una CBV como una vista.
  - `path('producto/<int:pk>/', ...)`: Define una URL con un parámetro dinámico (`<int:pk>`). `pk` significa "primary key" (clave primaria) y `int:` asegura que sea un entero. Esto permite URLs como `/producto/1/` para ver el producto con ID 1.

4. **Crea las nuevas plantillas CRUD:** Cada CBV necesita una plantilla específica para renderizar la interfaz de usuario. Todas estas plantillas heredarán de `base.html` y rellenarán el bloque `content`.

   - **`productos/templates/productos/producto_form.html`** (para Crear y Editar Producto):

```
{% extends 'productos/base.html' %}
{% block page_title %}{% if object %}Editar Producto{% else %}Crear Nuevo Producto{

{% block content %}
<div class="row">
    <div class="col-md-8 offset-md-2">
        <div class="card">
            <div class="card-header">
                <h2 class="card-title">{% if object %}Editar Producto{% else %}Crea
            </div>
            <div class="card-body">
                <form method="post" enctype="multipart/form-data"> {# enctype neces
                    {% csrf_token %} {# Token de seguridad requerido por Django #}
                    {{ form.as_p }} {# Renderiza el formulario como párrafos #}
                    <hr>
                    <button type="submit" class="btn btn-success">Guardar</button>
                    <a href="{% url 'productos:lista_productos' %}" class="btn btn-
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

   - **`productos/templates/productos/producto_detalle.html`**:

```
{% extends 'productos/base.html' %}
{% block page_title %}Detalle de Producto{% endblock %}

{% block content %}
<div class="card">
    <div class="card-header">
        <h2 class="card-title">Detalle del Producto</h2>
    </div>
    <div class="card-body">
        {% if object.imagen %}
            <img src="{{ object.imagen.url }}" alt="{{ object.nombre }}" class="img
        {% endif %}
        <h3>{{ object.nombre }}</h3>
        <p><strong>Categoría:</strong> {{ object.categoria.nombre }}</p>
```

```
        <p><strong>Precio:</strong> ${{ object.precio }}</p>
        <p><strong>Stock:</strong> {{ object.stock }} unidades</p>
        <hr>
        <h5>Embarques Asociados</h5>
        <ul>
        {% for embarque in object.embarques.all %}
            <li>{{ embarque.codigo_rastreo }} (Llegada: {{ embarque.fecha_llegada }
        {% empty %}
            <li>No hay embarques asociados.</li>
        {% endfor %}
        </ul>
    </div>
    <div class="card-footer">
        <a href="{% url 'productos:lista_productos' %}" class="btn btn-secondary">V
        <a href="{% url 'productos:producto_editar' object.pk %}" class="btn btn-pr
    </div>
</div>
{% endblock %}
```

- **productos/templates/productos/producto_confirm_delete.html**:

```
{% extends 'productos/base.html' %}
{% block page_title %}Confirmar Borrado{% endblock %}

{% block content %}
<div class="row">
    <div class="col-md-8 offset-md-2">
        <div class="card border-danger">
            <div class="card-header bg-danger text-white">
                <h2 class="card-title">Confirmar Borrado</h2>
            </div>
            <div class="card-body">
                <p>¿Estás seguro de que quieres eliminar el producto "<strong>{{ ob
                <p class="text-danger">Esta acción no se puede deshacer.</p>
                <form method="post">
                    {% csrf_token %}
                    <button type="submit" class="btn btn-danger">Confirmar Borrado<
                    <a href="{% url 'productos:producto_detalle' object.pk %}" clas
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

- **productos/templates/productos/embarque_lista.html**:

```
{% extends 'productos/base.html' %}
{% block page_title %}Listado de Embarques{% endblock %}

{% block content %}
<div class="table-responsive">
    <table class="table table-striped table-sm">
        <thead>
            <tr>
                <th scope="col">Código de Rastreo</th>
                <th scope="col">Fecha de Llegada</th>
                <th scope="col">Productos</th>
                <th scope="col">Acciones</th>
            </tr>
        </thead>
        <tbody>
```

```
            {% for embarque in embarques %}
            <tr>
                <td>{{ embarque.codigo_rastreo }}</td>
                <td>{{ embarque.fecha_llegada }}</td>
                <td>{{ embarque.productos.count }}</td>
                <td>
                    <div class="btn-group btn-group-sm">
                        <a href="{% url 'productos:embarque_detalle' embarque.pk %}
                        <a href="{% url 'productos:embarque_editar' embarque.pk %}"
                        <a href="{% url 'productos:embarque_eliminar' embarque.pk %
                    </div>
                </td>
            </tr>
            {% endfor %}
        </tbody>
    </table>
</div>
{% endblock %}
```

- **productos/templates/productos/embarque_detalle.html**:

```
{% extends 'productos/base.html' %}
{% block page_title %}Detalle de Embarque{% endblock %}

{% block content %}
<div class="card">
    <div class="card-header">
        <h2 class="card-title">Detalle del Embarque</h2>
    </div>
    <div class="card-body">
        <h3>{{ object.codigo_rastreo }}</h3>
        <p><strong>Fecha de Llegada:</strong> {{ object.fecha_llegada }}</p>
        <hr>
        <h5>Productos en este embarque</h5>
        <ul class="list-group">
        {% for producto in object.productos.all %}
            <li class="list-group-item">{{ producto.nombre }} (Stock: {{ producto.s
        {% empty %}
            <li class="list-group-item">No hay productos asociados a este embarque.
        {% endfor %}
        </ul>
    </div>
    <div class="card-footer">
        <a href="{% url 'productos:embarque_lista' %}" class="btn btn-secondary">Vo
        <a href="{% url 'productos:embarque_editar' object.pk %}" class="btn btn-pr
    </div>
</div>
{% endblock %}
```

- **productos/templates/productos/embarque_form.html**:

```
{% extends 'productos/base.html' %}
{% block page_title %}{% if object %}Editar Embarque{% else %}Crear Nuevo Embarque{

{% block content %}
<div class="row">
    <div class="col-md-8 offset-md-2">
        <div class="card">
            <div class="card-header">
                <h2 class="card-title">{% if object %}Editar Embarque{% else %}Crea
            </div>
            <div class="card-body">
```

```
                    <form method="post">
                        {% csrf_token %}
                        {{ form.as_p }}
                        <hr>
                        <button type="submit" class="btn btn-success">Guardar</button>
                        <a href="{% url 'productos:embarque_lista' %}" class="btn btn-s
                    </form>
                </div>
            </div>
        </div>
    </div>
    {% endblock %}
```

- **productos/templates/productos/embarque_confirm_delete.html**:

```
{% extends 'productos/base.html' %}
{% block page_title %}Confirmar Borrado de Embarque{% endblock %}

{% block content %}
<div class="row">
    <div class="col-md-8 offset-md-2">
        <div class="card border-danger">
            <div class="card-header bg-danger text-white">
                <h2 class="card-title">Confirmar Borrado de Embarque</h2>
            </div>
            <div class="card-body">
                <p>¿Estás seguro de que quieres eliminar el embarque "<strong>{{ ob
                <p class="text-danger">Esta acción no se puede deshacer.</p>
                <form method="post">
                    {% csrf_token %}
                    <button type="submit" class="btn btn-danger">Confirmar Borrado<
                    <a href="{% url 'productos:embarque_detalle' object.pk %}" clas
                </form>
            </div>
        </div>
    </div>
</div>
{% endblock %}
```

- **Explicación general de las plantillas CRUD:**
  - Todas heredan de `base.html` y sobrescriben `page_title` y `content`.
  - **Formularios (`_form.html`):** Contienen un `<form>` con `method="post"` (y `enctype="multipart/form-data"` si hay subida de archivos como imágenes). `{% csrf_token %}` es un token de seguridad esencial. `{{ form.as_p }}` renderiza todos los campos del formulario como párrafos.
  - **Detalle (`_detalle.html`):** Muestra los atributos del objeto (`{{ object.nombre }}`) y los objetos relacionados.
  - **Confirmar Borrado (`_confirm_delete.html`):** Una página simple que pide confirmación antes de eliminar, con un formulario POST para ejecutar el borrado y un botón de cancelar.

Paso 10: Creación de Datos de Ejemplo (Migraciones de Datos)

Para facilitar las pruebas y demostraciones, podemos poblar nuestra base de datos con datos de ejemplo utilizando migraciones de datos. Esto es más robusto que usar scripts únicos, ya que el historial de la base de datos queda registrado.

1. **Crea una migración vacía para los datos:**

```
python manage.py makemigrations --empty productos
```

   - **Explicación:** Este comando crea un nuevo archivo de migración en `productos/migrations/` (ej. `0002_auto_YYYYMMDD_HHMM.py`), pero sin operaciones generadas automáticamente. Lo rellenaremos manualmente con código Python para manipular los datos.

2. **Edita el archivo de migración generado** (`productos/migrations/0002_auto_YYYYMMDD_HHMM.py`) para añadir los datos de ejemplo: Abre el archivo de migración recién creado y reemplaza su contenido con este código.

```
# productos/migrations/0002_auto_20260131_0121.py
# (El nombre del archivo puede variar, pero el contenido será similar)

from django.db import migrations
from datetime import date # Importa para usar fechas

def crear_datos_ejemplo(apps, schema_editor):
    """Crea datos de ejemplo para los modelos Categoria, Producto y Embarque."""
    # apps.get_model permite acceder a los modelos en el estado de la migración actual
    Categoria = apps.get_model('productos', 'Categoria')
    Producto = apps.get_model('productos', 'Producto')
    Embarque = apps.get_model('productos', 'Embarque')

    # --- Crear Categorías ---
    print("Creando categorías de ejemplo...")
    cat_electronica = Categoria.objects.create(nombre='Electrónica', descripcion='Disposit
    cat_ropa = Categoria.objects.create(nombre='Ropa', descripcion='Prendas de vestir y ac
    cat_hogar = Categoria.objects.create(nombre='Hogar', descripcion='Artículos para el ho

    # --- Crear Productos ---
    print("Creando productos de ejemplo...")
    prod_laptop = Producto.objects.create(
        nombre='Laptop Pro X',
        precio=1200.50,
        stock=15,
        categoria=cat_electronica # Asocia a la categoría Electrónica
    )
    prod_smartphone = Producto.objects.create(
        nombre='Smartphone Z1',
        precio=850.00,
        stock=30,
        categoria=cat_electronica
    )
    prod_camiseta = Producto.objects.create(
        nombre='Camiseta de Algodón',
        precio=25.99,
        stock=100,
        categoria=cat_ropa
    )
    prod_silla = Producto.objects.create(
        nombre='Silla de Oficina Ergonómica',
        precio=150.75,
        stock=20,
        categoria=cat_hogar
    )
```

```
    # --- Crear Embarques y asociar productos (relación Muchos a Muchos) ---
    print("Creando embarques de ejemplo y asociando productos...")
    embarque1 = Embarque.objects.create(codigo_rastreo='TRACK-US-2026-001', fecha_llegada=
    embarque1.productos.set([prod_laptop, prod_smartphone]) # Asocia varios productos a u

    embarque2 = Embarque.objects.create(codigo_rastreo='TRACK-CN-2026-002', fecha_llegada=
    embarque2.productos.set([prod_camiseta, prod_silla])

    embarque3 = Embarque.objects.create(codigo_rastreo='TRACK-EU-2026-003', fecha_llegada=
    embarque3.productos.set([prod_laptop, prod_silla]) # Un producto puede estar en vario

def borrar_datos_ejemplo(apps, schema_editor):
    """Función inversa para eliminar los datos de ejemplo creados, si se deshace la migra
    Categoria = apps.get_model('productos', 'Categoria')
    Producto = apps.get_model('productos', 'Producto')
    Embarque = apps.get_model('productos', 'Embarque')

    # Elimina los registros creados. Usamos filtros para asegurar que solo borramos los da
    Embarque.objects.filter(codigo_rastreo__startswith='TRACK-').delete()
    Producto.objects.filter(nombre__in=['Laptop Pro X', 'Smartphone Z1', 'Camiseta de Alg
    Categoria.objects.filter(nombre__in=['Electrónica', 'Ropa', 'Hogar']).delete()


class Migration(migrations.Migration):

    dependencies = [
        ('productos', '0001_initial'), # Asegura que esta migración se ejecuta después de
    ]

    operations = [
        # Ejecuta la función crear_datos_ejemplo cuando se aplica la migración, y borrar_
        migrations.RunPython(crear_datos_ejemplo, reverse_code=borrar_datos_ejemplo),
    ]
```

- **Explicación:**
  - `apps.get_model('appname', 'ModelName')`: Permite acceder al modelo en el estado en que se encontraba durante la ejecución de esta migración. Es importante usar esto en lugar de importar los modelos directamente (`from .models import ...`) para evitar problemas con cambios futuros en los modelos.
  - `objects.create()`: Método para crear nuevas instancias de un modelo y guardarlas en la base de datos.
  - `productos.set([...])`: Para `ManyToManyField`, el método `set()` asigna la lista de objetos relacionados.
  - `reverse_code=borrar_datos_ejemplo`: Especifica la función que se debe ejecutar si se deshace (`migrate --rollback`) esta migración.
- **Generación de imágenes de ejemplo:** Crearemos otra migración de datos para generar imágenes placeholder para los productos.

```
python manage.py makemigrations --empty productos
```

- **Explicación:** Genera un nuevo archivo de migración vacío (ej. `0003_auto_YYYYMMDD_HHMM.py`).

Edita este nuevo archivo (`productos/migrations/0003_auto_YYYYMMDD_HHMM.py`) con el siguiente

código:

```
# productos/migrations/0003_auto_20260131_0126.py
# (El nombre del archivo puede variar)

from django.db import migrations
from django.conf import settings # Para acceder a MEDIA_ROOT
import os
from PIL import Image, ImageDraw, ImageFont # Librerías de Pillow

def crear_imagenes_ejemplo(apps, schema_editor):
    """Genera imágenes de placeholder para los productos de ejemplo."""
    Producto = apps.get_model('productos', 'Producto')

    # --- Configuración del directorio de imágenes ---
    media_dir = os.path.join(settings.MEDIA_ROOT, 'productos')
    os.makedirs(media_dir, exist_ok=True) # Crea la carpeta si no existe

    # --- Configuración de la fuente para el texto en la imagen ---
    try:
        font = ImageFont.truetype("arial.ttf", 24) # Intenta cargar una fuente común
    except IOError:
        font = ImageFont.load_default() # Si no la encuentra, usa la fuente por defecto

    # --- Datos de los productos para generar imágenes ---
    productos_data = {
        'Laptop Pro X': {'color': '#4a4a4a', 'file': 'laptop.png'},
        'Smartphone Z1': {'color': '#5a5a5a', 'file': 'smartphone.png'},
        'Camiseta de Algodón': {'color': '#6a6a6a', 'file': 'camiseta.png'},
        'Silla de Oficina Ergonómica': {'color': '#7a7a7a', 'file': 'silla.png'},
    }

    # --- Generar y guardar imágenes para cada producto ---
    for nombre_producto, data in productos_data.items():
        try:
            producto = Producto.objects.get(nombre=nombre_producto)
            img_path = os.path.join(media_dir, data['file'])

            # Crea una imagen en blanco con el color especificado
            img = Image.new('RGB', (600, 400), color=data['color'])
            d = ImageDraw.Draw(img)

            # Dibuja el nombre del producto en el centro de la imagen
            text = f"{producto.nombre}\n(Placeholder)"
            d.text((300, 200), text, fill=(255, 255, 255), font=font, anchor="mm")

            # Guarda la imagen en la ruta especificada
            img.save(img_path)

            # Actualiza el campo 'imagen' del producto en la base de datos
            producto.imagen = os.path.join('productos', data['file']) # Ruta relativa a MI
            producto.save()

        except Producto.DoesNotExist:
            # Si el producto no existe (ej. fue borrado), ignora y continúa
            continue

def borrar_imagenes_ejemplo(apps, schema_editor):
    """Elimina las imágenes de placeholder generadas y limpia los campos en la BD."""
    Producto = apps.get_model('productos', 'Producto')
    productos_data = { # Mismos datos para identificar las imágenes a borrar
        'Laptop Pro X': {'file': 'laptop.png'},
```

```
            'Smartphone Z1': {'file': 'smartphone.png'},
            'Camiseta de Algodón': {'file': 'camiseta.png'},
            'Silla de Oficina Ergonómica': {'file': 'silla.png'},
        }

        for nombre_producto, data in productos_data.items():
            try:
                producto = Producto.objects.get(nombre=nombre_producto)
                if producto.imagen:
                    # Construye la ruta absoluta del archivo de imagen
                    img_path = os.path.join(settings.MEDIA_ROOT, producto.imagen.name)
                    if os.path.exists(img_path):
                        os.remove(img_path) # Borra el archivo físico del disco

                    # Limpia el campo 'imagen' en el modelo del producto
                    producto.imagen = None
                    producto.save()
            except Producto.DoesNotExist:
                continue

    class Migration(migrations.Migration):

        dependencies = [
            ('productos', '0002_auto_20260131_0121'), # Asegura que esta migración se ejecuta
        ]

        operations = [
            migrations.RunPython(crear_imagenes_ejemplo, reverse_code=borrar_imagenes_ejemplo)
        ]
```

- **Explicación:**
  - `PIL.Image`, `PIL.ImageDraw`, `PIL.ImageFont`: Clases de la librería Pillow para manipulación de imágenes.
  - `os.makedirs(media_dir, exist_ok=True)`: Crea el directorio donde se guardarán las imágenes si aún no existe.
  - `Image.new('RGB', (width, height), color)`: Crea una nueva imagen con el modo de color, tamaño y color de fondo especificados.
  - `ImageDraw.Draw(img)`: Crea un objeto `Draw` para dibujar en la imagen.
  - `d.text((x, y), text, fill, font, anchor)`: Dibuja texto en la imagen. `anchor="mm"` centra el texto.
  - `img.save(img_path)`: Guarda la imagen generada.
  - `producto.imagen = os.path.join('productos', data['file'])`: Asigna la ruta relativa de la imagen al campo `imagen` del modelo.

3. **Aplica las migraciones (ambas):** Una vez que hayas modificado ambos archivos de migración, aplica los cambios a la base de datos.

```
python manage.py migrate
```

Conclusión

¡Felicidades! Has construido una aplicación Django completa con funcionalidades CRUD para dos modelos relacionados, una interfaz de administración mejorada y una interfaz pública funcional.

Para ver el resultado final, simplemente ejecuta el servidor:

```
python manage.py runserver
```

Y visita `http://127.0.0.1:8000/` en tu navegador.

Este proyecto sienta las bases para funcionalidades mucho más complejas. ¡Sigue explorando y construyendo!