Refactorización de una Aplicación Django con Diseño Orientado a Objetos

Trabajo Práctico Integrador

Profesor: Gianluca Piriz

Materia: Programación Avanzada

Integrantes: Nahuel Dalesio, Darío Giménez y Federico Paál

LA ANTIGUA REVISTERÍA

Índice

Motivación

De un Prototipo Funcional a una Aplicación Profesional

Nuestro objetivo es aplicar los conceptos de la materia para transformar un código repetitivo y difícil de mantener en una solución robusta y escalable.

El Problema:

Código Duplicado en Modelos

Violación del Principio DRY (Don't Repeat Yourself)

```python
class Producto(models.Model):
    titulo = models.CharField(max_length=40)
    autor = models.CharField(max_length=40)
    precio = models.IntegerField()
    imagen = models.ImageField(null=True, blank=True, upload_to='Productos')
    creacion = models.DateField(null=True, blank=True)
    categoria = models.CharField(max_length=40)
    texto = models.CharField(max_length=500, null=True, blank=True)

    prox_categoria = models.CharField(max_length=40, null=True, blank=True)
```

```python
def ver_detalles(req, producto_id):

    producto = Producto.objects.get(id=producto_id)
    if producto.categoria == "LIBROS":
        return render(req, "detallesLibro.html", {"libro": producto})

    elif producto.categoria == "NOVEDADES":
        return render(req, "detallesNovedad.html", {"novedad": producto})

    elif producto.categoria == "MERCHANDISINGS":
        return render(req, "detallesMerchandising.html", {"merch": producto})
```

Teníamos una misma clase para categorizarla en distintas lo cual es una mala práctica debido a que hay baja cohesión y un alto acoplamiento.

Solución con Herencia

Abstracción y Reutilización con Herencia

Creamos una clase base abstracta Producto que centraliza toda la lógica común.

```python
class Producto(models.Model):
    CATEGORIA_CHOICES = (
        ('libro', 'Libro'),
        ('merchandising', 'Merchandising'),
    )
    titulo = models.CharField(max_length=100)
    precio = models.DecimalField(max_digits=10, decimal_places=2)
    imagen = models.ImageField(upload_to='productos/', null=True, blank=True)
    categoria = models.CharField(max_length=20, choices=CATEGORIA_CHOICES, editable=False)
    texto = models.TextField(verbose_name="Descripción", null=True, blank=True)
    creacion = models.DateField(auto_now_add=True, null=True, blank=True)
    es_novedad = models.BooleanField(default=False, verbose_name="¿Es novedad?")

    creacion = models.DateField(auto_now_add=True, null=True, blank=True)

    class Meta:
        abstract = True
        ordering = ['-creacion']

    def __str__(self):
        return self.titulo

# --- MODELOS CONCRETOS QUE HEREDAN DE PRODUCTO ---
class Libro(Producto):
    autor = models.CharField(max_length=100, null=True, blank=True)

    def save(self, *args, **kwargs):
        self.categoria = 'libro'
        super().save(*args, **kwargs)

    class Meta:
        verbose_name = "Libro"
        verbose_name_plural = "Libros"

class Merchandising(Producto):
    def save(self, *args, **kwargs):
        self.categoria = 'merchandising'
        super().save(*args, **kwargs)

    class Meta:
        verbose_name = "Merchandising"
        verbose_name_plural = "Merchandising"
```

El Problema: Vistas y URLs Repetitivas

Lógica Dispersa y Falta de Flexibilidad

En urls.py

```python
#Listas Productos
path('_libros/', libros, name="_Libros"),
path('_merchs/', merchs, name="_Merchs"),
path('_novedades/', novedades, name="_Novedades"),
```

En views.py

```python
def libros(req):
    libros = Producto.objects.all().filter(categoria="LIBROS").order_by("id").reverse()
    return render(req, "libros.html", {"libros": libros})

def novedades(req):
    novedades = Producto.objects.all().filter(categoria="NOVEDADES").order_by("id").reverse()
    return render(req, "novedades.html", {"novedades": novedades})

def merchs(req):
    merchs = Producto.objects.all().filter(categoria="MERCHANDISINGS").order_by("id").reverse()
    return render(req, "merchandisings.html", {"merchs": merchs})
```

Una función y una URL para cada tipo de producto. Para añadir una nueva categoría, había que duplicar todo de nuevo.

Solución con Polimorfismo y CBV

Una Vista para Gobernarlos a Todos

En urls.py

```
# Productos (Rutas unificadas)
path('productos/<str:categoria>/', views.ProductoListView.as_view(), name='producto_list'),
path('producto/<int:pk>/', views.ProductoDetailView.as_view(), name='producto_detail'),
```

En views.py

```
class ProductoListView(ListView):
    template_name = 'tienda/producto_list.html'
    context_object_name = 'productos'

    def get_queryset(self):
        categoria = self.kwargs.get('categoria')
        if categoria == 'libros':
            self.model = Libro
            return Libro.objects.all()
        elif categoria == 'merchandising':
            self.model = Merchandising
            return Merchandising.objects.all()
        return []

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['categoria_titulo'] = self.kwargs.get('categoria').replace('-', ' ').title()
        return context
```

Ahora, una única Vista Basada en Clase (CBV) maneja polimórficamente la solicitud. La vista se adapta según el parámetro de la URL, consultando el modelo correcto. Esto es escalable.

Encapsulamiento del Carrito

Agrupando Datos y Comportamiento

Toda la lógica del carrito está encapsulada en su propia clase. Las vistas no saben cómo funciona, solo qué puede hacer. Esto es encapsulamiento

Reutilización de Código con Mixins

Control de Acceso Elegante

```python
class StaffRequiredMixin(UserPassesTestMixin):
    """Mixin para requerir que el usuario sea staff."""
    def test_func(self):
        return self.request.user.is_authenticated and hasattr(self.request.user, 'perfil') and self.request.user.perfil.rol == 'staff'
```

```python
class ProductoCreateView(StaffRequiredMixin, CreateView):
    template_name = 'tienda/producto_form.html'
    success_url = reverse_lazy('tienda:index')

    def get_form_class(self):
        categoria = self.kwargs.get('categoria')
        return LibroForm if categoria == 'libro' else MerchandisingForm

    def form_valid(self, form):
        messages.success(self.request, 'Producto creado con éxito.')
        return super().form_valid(form)

    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['titulo_form'] = "Crear Nuevo Producto"
        return context
```

En lugar de repetir if user.is_staff en cada vista, creamos un Mixin que se puede 'mezclar' con cualquier vista para añadir la funcionalidad de autorización. Esto es composición sobre herencia.

Patrones de Diseño en Acción

Aplicando Soluciones Probadas

Template Method

```python
class NovedadesListView(ListView):

    # Personalizamos el PASO 1: ¿De dónde sacamos los datos?
    def get_queryset(self):
        libros = Libro.objects.filter(es_novedad=True)
        merch = Merchandising.objects.filter(es_novedad=True)
        return sorted(chain(libros, merch), ...)

    # Personalizamos el PASO 2: ¿Qué datos extra enviamos?
    def get_context_data(self, **kwargs):
        context = super().get_context_data(**kwargs)
        context['categoria_titulo'] = "Nuestras Novedades"
        return context
```

Factory

```python
class ProductoCreateView(StaffRequiredMixin, CreateView):

    # Este método es la "fábrica" de formularios.
    def get_form_class(self):
        categoria = self.kwargs.get('categoria')

        if categoria == 'libro':
            # Decide crear un LibroForm
            return LibroForm
        else:
            # Decide crear un MerchandisingForm
            return MerchandisingForm
```

Singleton

```python
class Carrito:
    def __init__(self, request):
        # 1. Obtenemos el punto de acceso global: la sesión
        self.session = request.session

        # 2. Buscamos la "única instancia" de datos del carrito
        carrito = self.session.get("carrito")

        # 3. Si no existe, la creamos por primera y única vez
        if not carrito:
            carrito = self.session["carrito"] = {}

        self.carrito = carrito
```

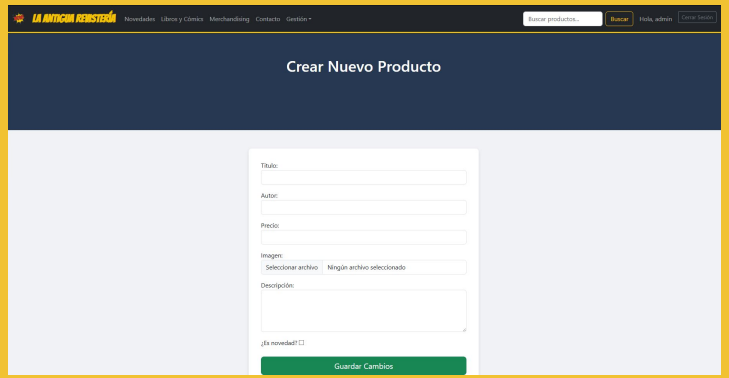
Usamos patrones de diseño que el propio Django promueve para escribir código limpio.

Demo en Vivo

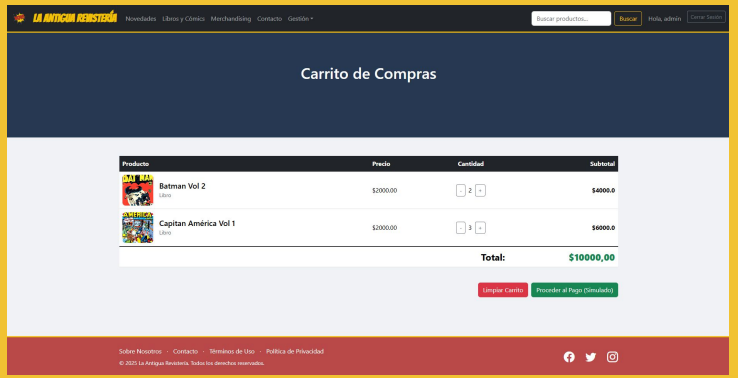
Momento de mostrar la Web App

La Web App fue hecha en el framework de Django, donde se podrá observar mediante pruebas en vivo el correcto funcionamiento de este.

Pantalla de Inicio



Creación de un producto



Carrito de Compras

Resultados y Beneficios Obtenidos

Comparativa: Antes vs Después

Características	Proyecto Original	Proyecto Refactorizado
Código	Repetitivo y extenso	DRY y conciso
Mantenimiento	Difícil y propenso a errores	Fácil y seguro
Escalabilidad	Baja	Alta
Diseño	Funcional	Orientado a Objetos

Conclusiones del Proyecto

Lecciones Aprendidas

- La POO y el DOO son esenciales para la calidad del software.
- La herencia y el polimorfismo reducen drásticamente la duplicación.
- Los patrones de diseño ofrecen soluciones elegantes a problemas recurrentes.
- Refactorizar nos enseñó a identificar 'code smells' (malos olores en el código).

Conclusiones de la Cursada

Sobre la cursada:

- La materia de Programación Avanzada nos permitió entender temas y conceptos para así adquirirlos y usarlos propiamente en el desarrollo de la cursada.
- Gracias al aprendizaje, se pudo implementar dichos conceptos en tareas, trabajos y hasta en el parcial.
- Además, Programación Avanzada también fue un complemento para otras materias de la tecnicatura.

```
1
2
3
4
5
```

```
Print(‘¡Gracias!’)
```