

ISS-09 — Catálogo con ModelViewSet

Publicación de los recursos ProductType y Product utilizando Herencia de ModelViewSet en DRF

StoreLab REST API — Material Pedagógico de Ingeniería de Sistemas

¿Qué construiremos en el ISS-09?

□ Objetivo General del ISS-09

- **Publicación de Recursos Homogéneos:**

Exponer los endpoints CRUD para los modelos `ProductType` y `Product` siguiendo las mismas reglas fijadas en clientes (ISS-08).

- **Abstracción de ViewSets (CatalogViewSet):**

Evitar la duplicación de código extrayendo el comportamiento común (`AllowAny`, baja lógica `perform\_destroy` y filtrado `?status=`) a un ViewSet padre.

- **Optimización de Consultas ORM:**

Implementar `select\_related('product\_type')` en el ViewSet de productos para evitar el problema N+1 de SQL.

- **Aislamiento de Rutas con SimpleRouter:**

Publicar `/api/product-types/` y `/api/products/` de forma independiente.

□ Reglas de Negocio y Borde HTTP

- **Modo de Acceso OPEN (Fase I):**

Las peticiones POST, GET, PATCH y DELETE no exigen token JWT ni permisos en esta fase (`permission\_classes = [AllowAny]`).

- **Estructura camelCase:**

El cliente HTTP envía `productTypeld` y recibe respuestas JSON formateadas en `camelCase` por `django-rest-framework-camel-case`.

- **Baja Lógica Garantizada:**

El método `DELETE` ejecuta una baja lógica cambiando `status = RecordStatus.INACTIVE` mediante `perform\_destroy`.

- **Protección de Integridad Catálogo:**

Si un `ProductType` tiene productos asociados, el borrado físico directa en BD lanza `RestrictedError` (`on\_delete=RESTRICT`).

Punto de Partida del Proyecto (Post ISS-08)

Estado del Sistema al iniciar el ISS-09

✓ Modelos de Catálogo Existentes (ISS-05):

Clases ``ProductType`` y ``Product`` definidas en ``apps/product/models.py`` con sus campos, validadores (``MinLengthValidator``, ``MinValueValidator``) y restricciones de integridad (``on_delete=RESTRICT``).

✓ Serializers de Catálogo Listos (ISS-07):

``ProductTypeSerializer`` y ``ProductSerializer`` creados en ``apps/product/serializers.py`` con mapeo de ``product_type_id`` y ``product_type_name``.

✓ Patrón de CRUD Probado en Clientes (ISS-08):

``ClientViewSet`` con ``ModelViewSet``, ``SimpleRouter`` y ``perform_destroy`` implementado y verificado en tests.

✓ Middlewares y Renderers Configurados (ISS-07):

``CorsMiddleware`` y ``CamelCaseJSONRenderer`` operando a nivel global de ``settings.py``.

✓ Aclaración Crucial sobre el ISS-10:

El endpoint ``/api/products/available/`` (ListAPIView para catálogo activo/disponible) AÚN NO EXISTE. Se construirá e integrará en el ISS-10.

Construcción de Vistas: apps/product/views.py

```
from rest_framework.permissions import AllowAny
from apps.common.status import RecordStatus
from apps.product.models import Product, ProductType
from apps.product.serializers import ProductSerializer, ProductTypeSerializer

# --- CLASE BASE ABSTRACTA DE CATÁLOGO ---
class CatalogViewSet(viewsets.ModelViewSet):
    permission_classes = [AllowAny]
    lookup_value_converter = "int"

    def get_queryset(self):
        queryset = super().get_queryset()
        status_value = self.request.query_params.get("status")
        if status_value:
            queryset = queryset.filter(status=status_value)
        return queryset

    def perform_destroy(self, instance):
        instance.status = RecordStatus.INACTIVE
        instance.save(update_fields=["status", "updated_at"])

# --- VIEWSETS CONCRETOS ---
class ProductTypeViewSet(CatalogViewSet):
    queryset = ProductType.objects.all()
    serializer_class = ProductTypeSerializer

class ProductViewSet(CatalogViewSet):
    queryset = Product.objects.select_related("product_type")
    serializer_class = ProductSerializer
```

Análisis Semántico de la Implementación en Views

Bloque 1: Clase Base CatalogViewSet

Hereda de `viewsets.ModelViewSet`. Agrupa la configuración común para todos los recursos del catálogo. Define `permission_classes = [AllowAny]` para la Fase I y fuerza los identificadores numéricos con `lookup_value_converter = 'int'`.

Bloque 2: Filtrado Dinámico (`get_queryset`)

Extiende la consulta base para inspeccionar los parámetros de la URL (`request.query_params.get('status')`). Permite filtrar dinámicamente `/api/products/?status=active` o inactivos.

Bloque 3: Baja Lógica (`perform_destroy`)

Intercepta la acción HTTP `DELETE`. En lugar de ejecutar `instance.delete()` físico en BD, cambia el atributo `status` a `RecordStatus.INACTIVE` y guarda especificando `update_fields`.

Bloque 4: Optimización `select_related`

`ProductViewSet` declara `get_queryset = Product.objects.select_related('product_type')`. Realiza un JOIN SQL automático que trae el nombre del tipo sin lanzar peticiones adicionales por cada producto.

Enrutamiento Modular con SimpleRouter

```

# apps/product/urls.py
from rest_framework.routers import SimpleRouter
from apps.product.views import (
    ProductTypeViewSet, ProductViewSet
)

router = SimpleRouter(use_regex_path=False)
router.register("product-types",
                ProductTypeViewSet,
                basename="product-type")
router.register("products",
                ProductViewSet,
                basename="product")

urlpatterns = router.urls

# config/urls.py (Inclusión global)
urlpatterns = [
    path("admin/", admin.site.urls),
    path("api/", include("apps.client.urls")),
    path("api/", include("apps.product.urls")),
]

```

Endpoints de API Expuestos

- **GET /api/product-types/**
Listar tipos de producto (filtro ?status=).
- **POST /api/product-types/**
Crear nuevo tipo de producto.
- **GET /api/product-types/<int:pk>/**
Detalle de un tipo de producto.
- **PUT/PATCH /api/product-types/<int:pk>/**
Actualización completa o parcial.
- **DELETE /api/product-types/<int:pk>/**
Baja lógica (status=inactive).
- **GET /api/products/**
Listar productos con productTypeName incorporado.
- **POST /api/products/**
Crear producto (acepta productTypeId).
- **GET /api/products/<int:pk>/**
Detalle completo de un producto.
- **PUT/PATCH /api/products/<int:pk>/**
Actualización parcial (ej: stock).
- **DELETE /api/products/<int:pk>/**
Baja lógica del producto.

Verificación con Pruebas de Integración (APITestCase)

□ Suite de Pruebas ProductApiTests en Acceso OPEN

► **setUp(self):**

Crea directamente en ORM el `ProductType` de apoyo ('Bebida', status=ACTIVE). No autentica ni usa tokens Bearer en esta fase.

► **test_product_type_crud():**

Ejecuta el ciclo de vida completo de `ProductType`: POST (201, default inactive) → GET list (200) → GET detail (200) → PATCH (200, activa) → DELETE (204, verifica baja lógica en BD con status=inactive).

► **test_product_retrieve_update_and_soft_delete():**

Petición POST con JSON camelCase (`productType`, `price`, `stock`, `status`) → GET detail → PATCH (actualiza stock a 9) → DELETE (204, confirma baja lógica).

► **test_default_status_and_type_restriction():**

Verifica que el alta sin status deja `status='inactive'` y `stock=0`. Comprueba que al intentar eliminar en ORM el `ProductType` 'Bebida' con productos asociados, Django lanza `RestrictedError`.

► **test_duplicate_product_name():**

Envía dos peticiones POST con el mismo nombre 'Café'. La primera responde 201 y la segunda es rechazada con HTTP 400 Bad Request.

Criterios de Aceptación, Evidencias y Compuerta GATE

□ Criterios de Aceptación & Evidencias

AC-09-01:

Crear un producto activo con stock aparece en el catálogo (HTTP 201).

AC-09-02:

El nombre de producto duplicado responde HTTP 400 Bad Request.

AC-09-03:

Los valores por defecto son status=inactive y stock=0.

EVI-09-01:

Ejecución de `python manage.py test apps.product` con resultado OK (PASS) en PostgreSQL, MySQL y SQL Server.

AC	Verificación	Evidencia	Resultado
AC-09-01	POST producto	EVI-09-01	PASS
AC-09-02	Duplicado (400)	EVI-09-01	PASS
AC-09-03	Defaults inact/0	EVI-09-01	PASS

Preguntas de Evaluación y Defensa Oral

Preguntas para Defender la Construcción Técnica del ISS-09

❑ **1. ¿Por qué se optó por crear una clase base `CatalogViewSet` en lugar de definir individualmente `ProductTypeViewSet` y `ProductViewSet`?**

❑ Para reutilizar las decisiones transversales de arquitectura (``permission_classes = [AllowAny]``, ``lookup_value_converter = 'int'``, el filtrado dinámico por ``?status=`` en ``get_queryset`` y la baja lógica en ``perform_destroy``) sin duplicar código.

❑ **2. ¿Qué beneficio técnico aporta declarar `queryset = Product.objects.select_related('product_type')` en `ProductViewSet`?**

❑ Realiza un ``INNER JOIN`` a nivel de base de datos en la consulta inicial, evitando el problema N+1 de SQL cuando el serializer requiere incluir ``product_type_name``.

❑ **3. ¿Por qué el borrado de un `ProductType` con productos asociados mediante ORM genera una excepción `RestrictedError`?**

❑ Porque en el ISS-05 la clave foránea se definió con ``on_delete=models.RESTRICT``. Esto impide la eliminación del padre mientras existan registros hijos vinculados.

❑ **4. ¿Por qué la ruta `/api/products/available/` NO forma parte de este ISS?**

❑ Porque corresponde a una vista especializada de filtrado (``ListAPIView`` para catálogo disponible/activo) que se construye explícitamente en el ISS-10.