

CURSO DE ANÁLISIS Y DISEÑO CON DJANGO REST FRAMEWORK

# ISS-11 — Registro de venta con APIView

Transaccionalidad ACID, bloqueos pesimistas, congelamiento de precio histórico y actualización de stock en una sola operación atómica.

# ¿Qué construiremos en este ISS?

## Objetivo Principal

- **Operación Atómica:** Registrar la venta en un único paso transaccional.
- **Componentes en Juego:** Cliente, detalles de productos, precio histórico congelado, calculo de totales e inventario.
- **Garantía ACID:** Si cualquier validación o guardado falla, la transacción se revierte al 100%.
- **Cero Estado Corrupto:** No pueden existir ventas huérfanas sin stock ni desprendimientos parciales de inventario.

## Componentes Técnicos Clave

- **SaleQuerySet.register:** Invariante de negocio encapsulada dentro del manager/queryset del ORM.
- **transaction.atomic():** Context manager de Django que delimita la transacción de base de datos.
- **select\_for\_update():** Bloqueo pesimista de filas (row-level locking) en cliente y productos.
- **SaleRegisterSerializer:** Serializer no-ModelSerializer que orquesta la validación de entrada y la llamada al manager.
- **SaleRegisterAPIView:** Endpoint HTTP de tipo APIView en acceso OPEN (Fase I).

# Punto de Partida y Arquitectura Previas

---

## Estado del Proyecto (Superado ISS-10)

- **Modelo Sale y ProductSale (ISS-06):** Existencia previa de modelos sin lógica transaccional de registro en el manager.
- **Selección de Motor (ISS-01):** Compatibilidad comprobada con PostgreSQL, MySQL y SQL Server para bloqueos `select_for_update()`.
- **Catálogo e Inventario (ISS-05/09/10):** Estructura de productos con campo stock y filtrado de disponibles.
- **Serializers de Lectura (ISS-07):** ProductSaleSerializer y SaleSerializer listos para formatear la respuesta enriquecida en camelCase.
- **Fase Actual (Fase I):** Acceso libre (OPEN) con `permission_classes = [AllowAny]`. Sin autenticación JWT hasta la Fase II.

# Construcción: SaleQuerySet.register() — Parte 1

## Validaciones Previas e Invariantes

- **Sintaxis de Llamada:** `Sale.objects.register(client, sale_date, items, discounts)`
- **Validación de Items:** Suma y valida que items no esté vacío y que cada quantity sea  $\geq 1$ .
- **Consolidación de Cantidades:** Agrupa ítems repetidos en un diccionario `quantities[product_id]`.
- **Validación de Descuentos:** Verifica que discounts no sea negativo y redondea a 2 decimales.

## Bloqueo Pesimista (`select_for_update`)

- **Bloqueo del Cliente:**  
`Client.objects.select_for_update().get(pk=client.pk)`
- **Verificación de Cliente:** Lanza `ValidationError` si el cliente no está en estado active.
- **Bloqueo de Productos:**  
`Product.objects.select_for_update().select_related('product_type').filter(pk__in=quantities).order_by('id')`
- **Prevención de Deadlocks:** El ordenamiento explícito por ID (`.order_by('id')`) previene interbloqueos en la BD.

## Construcción: SaleQuerySet.register() — Parte 2

### Validación de Dominio y Creación

- **Existencia de Productos:** Verifica que el número de productos bloqueados coincida con los solicitados.
- **Estado de Producto y Tipo:** Exige status == ACTIVE tanto en el producto como en su ProductType.
- **Control de Stock:** Comprueba que product.stock >= quantity acumulada.
- **Instancia Venta Borrador:** Crea Sale con status=INACTIVE y totales en 0.00 como salvaguarda.

### Congelamiento de Precio y Totales

- **Precio Histórico:** unit\_price = product.price (congelado en ProductSale).
- **Total de Línea:** line\_total = (unit\_price \* quantity).quantize(0.01).
- **Cálculo de Subtotal y Total:** Acumula subtotal, verifica discounts <= subtotal, calcula total = subtotal + tax - discounts.
- **Activación y Descuento Stock:** Actualiza Sale a status=ACTIVE y descuenta product.stock -= quantity con update\_fields.

# Serializers y Vistas: SaleRegisterSerializer y APIView

---

## SaleRegisterSerializer

- **Estructura de Entrada:** client\_id (PrimaryKeyRelatedField), sale\_date, discounts, items (many=True).
- **Mapeo de Datos:** Transforma la entrada del cliente hacia la firma exigida por el manager.
- **Manejo de Excepciones:** Captura DjangoValidationError y la traduce a serializers.ValidationError (HTTP 400).

## SaleRegisterAPIView & Collection

- **Selección de APIView:** Uso de APIView por tratarse de una operación no genérica multi-recurso.
- **Respuesta Enriquecida:** Valida, ejecuta serializer.save() y retorna SaleSerializer con prefetch\_related('lines\_\_product').
- **Enrutamiento Explícito:** Ruta fija POST /api/sales/ mediante SaleCollectionAPIView (sin SimpleRouter).

# Flujo de Ejecución en la Base de Datos

## Secuencia Transaccional ACID

- **1. HTTP POST /api/sales/:** Envío de JSON con clientId, saleDate, discounts e items.
- **2. Validaciones de Forma:** DRF valida tipos de datos y formatos numéricos.
- **3. Apertura atomic():** Inicia la transacción de base de datos.
- **4. Bloqueo pesimista:** SELECT FOR UPDATE sobre cliente y productos ordenados.
- **5. Creación Venta Inactiva:** Garantía de ID sin visibilidad comercial.
- **6. Creación Detalles:** ProductSale creados con unit\_price congelado.
- **7. Cierre de Venta y Stock:** Totales calculados, estado ACTIVE, stock descontado.
- **8. COMMIT / ROLLBACK:** Exito -> COMMIT y HTTP 201 | Error -> ROLLBACK inmediato y HTTP 400.

# Errores Frecuentes de Arquitectura

---

## Fallas Comunes

- **Usar ModelAndViewSet.create:** Descompone el alta en peticiones separadas rompiendo la atomicidad.
- **Omitir select\_for\_update():** Permite condiciones de carrera sobre el inventario bajo alta concurrencia.
- **Bloquear sin Ordenar:** Provoca deadlocks en la BD cuando dos ventas bloquean los mismos productos en orden inverso.
- **Tomar Precio del Catálogo:** No congelar unit\_price alteraría ventas pasadas si Product.price cambia.

## Solución Aplicada

- **APIView Dedicada:** Encapsula la transacción completa en un solo endpoint.
- **Transacción Explícita:** transaction.atomic() garantiza consistencia 100%.
- **.order\_by('id'):** Garantía de orden estricto en el bloqueo pesimista.
- **Copia de unit\_price:** Guarda el valor del instante de venta en ProductSale.

# Crterios de Aceptación y Evidencias

## Crterios de Aceptación

- **AC-11-01:**  $2 \times 10.00 + 1 \times 4.50 - 1.50 = \text{total } 23.00$ , tax 0.00, status active.
- **AC-11-02:** Si Product.price cambia a 99.00, la venta conserva unit\_price = 10.00.
- **AC-11-03:** Stock insuficiente responde 400 y no crea la venta.
- **AC-11-04:** Fallo al guardar el stock ejecuta rollback completo (Sale=0, ProductSale=0).
- **AC-11-05:** Cliente inactivo es rechazado con HTTP 400.

## Evidencias y Verificación

- **Suite de Pruebas:** apps/sale/tests.py -> SaleTransactionTests
- **Test de Invariante:**  
test\_register\_snapshots\_price\_and\_discounts\_stock
- **Test de Rollback Simulado:**  
test\_stock\_failure\_rolls\_back\_the\_sale con unittest.mock.patch
- **Evidencia Registrada:** EVI-11-01: Cinco pruebas PASS en PostgreSQL, MySQL y SQL Server.
- **Resultado GATE:** PASS — Aprobación total de criterios de aceptación.

# Compuerta de Decisión (GATE)

## Matriz GATE — ISS-11

- **AC-11-01 | Totales matemáticos correctos:** Verificación: POST con ítems y descuento | Evidencia: EVI-11-01 | Resultado: PASS
- **AC-11-02 | Inmutabilidad de precio histórico:** Verificación: Cambio posterior en Product.price | Evidencia: EVI-11-01 | Resultado: PASS
- **AC-11-03 | Validación de stock suficiente:** Verificación: POST con cantidad > stock | Evidencia: EVI-11-01 | Resultado: PASS
- **AC-11-04 | Rollback ante error en stock:** Verificación: Mock de falla en Product.save | Evidencia: EVI-11-01 | Resultado: PASS
- **AC-11-05 | Rechazo de cliente inactivo:** Verificación: POST con client status=inactive | Evidencia: EVI-11-01 | Resultado: PASS

# Preguntas de Defensa Oral

## Evaluación de Comprensión Técnica

- **1. ¿Por qué se utilizó APIView en lugar de ModelAndViewSet para el registro de ventas?:** Porque el alta de una venta involucra múltiples recursos (Sale, ProductSale, Product) y reglas transaccionales no genéricas que un ModelAndViewSet estándar descompone de forma no atómica.
- **2. ¿Qué función cumple select\_for\_update() y por qué es obligatorio ordenar los productos por ID?:** Efectúa un bloqueo de filas pesimista en la BD para evitar condiciones de carrera en el stock. Ordenar por ID evita deadlocks (interbloqueos) cuando dos hilos intentan bloquear los mismos productos.
- **3. ¿Por qué el manager crea primero la Venta en estado INACTIVE antes de calcular los totales?:** Para obtener una clave primaria segura y vincular los detalles ProductSale. Si cualquier cálculo posterior o descuento de stock falla, transaction.atomic() deshace todo el bloque.
- **4. ¿Cómo se garantiza que un cambio posterior de precio en el catálogo no afecte ventas pasadas?:** El precio se copia explícitamente desde Product.price hacia ProductSale.unit\_price durante el registro, desvinculando la transacción del catálogo activo.