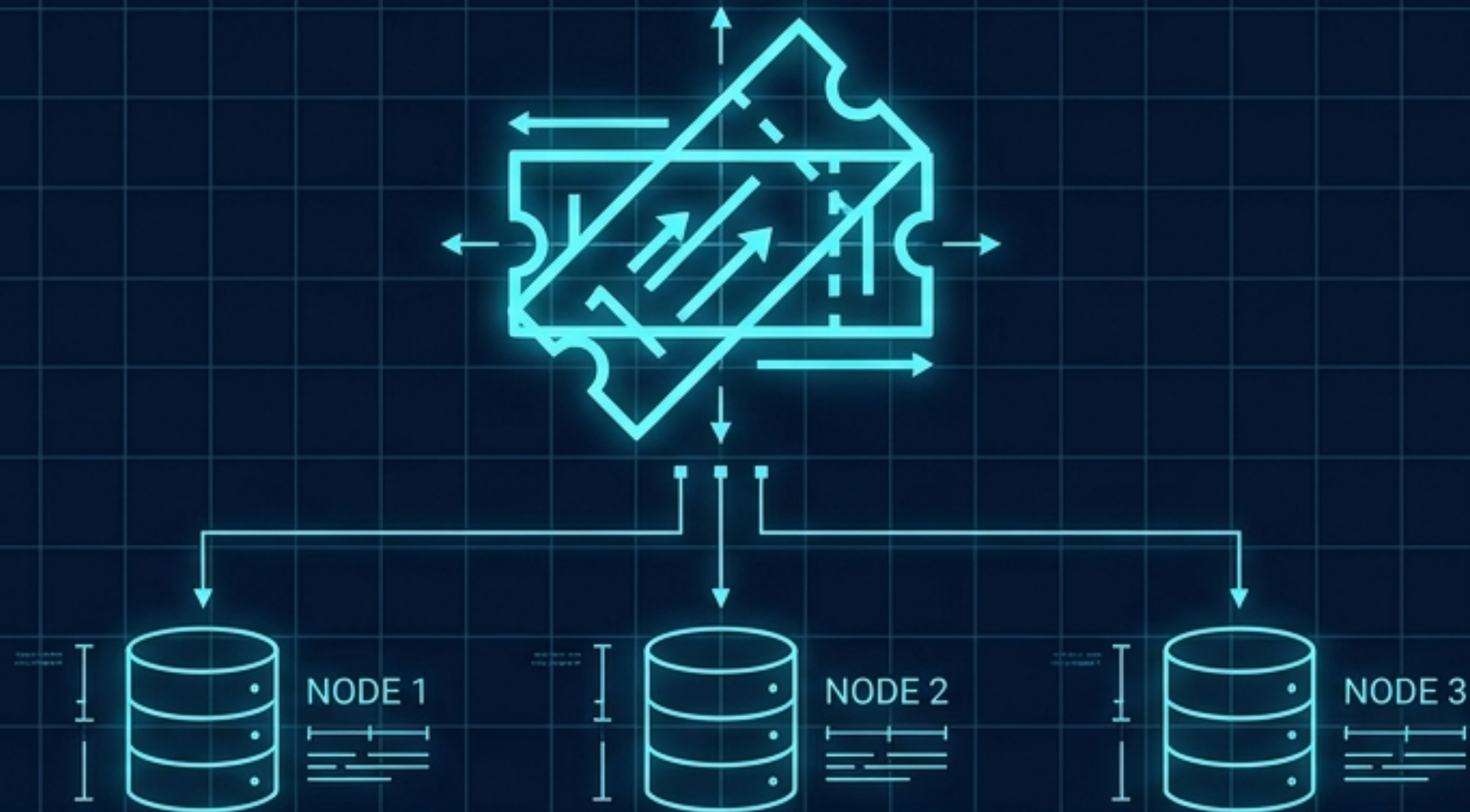


ISS-06 — Sale y ProductSale

Construcción del Modelo Transaccional de Ventas



Preservando la integridad histórica y la concurrencia en Django.

¿Qué construiremos? Evitando la trampa del M2M



Construiremos el modelo de cabecera (**Sale**) y su detalle de línea (**ProductSale**).
Nuestro objetivo principal: evitar el uso ingenuo de **ManyToManyField** para preservar los precios históricos y la integridad transaccional.

Punto de partida: ISS-05 Superado

`apps/product/models.py`



ProductType

Product

- El catálogo comercial ya existe.
- Modelos **ProductType** y **Product** están contruidos y validados.
- El negocio ya tiene '**qué**' vender. Ahora definiremos '**cómo**' venderlo.

Razón de Arquitectura: ¿Por qué NO usar M2M simple?

M2M Simple



Solo vincula ID-**Venta** con ID-Producto.

Falla Fatal: Si **Product.price** sube mañana, el total de las ventas del mes pasado cambia mágicamente. Se destruye el historial.
No hay dónde guardar la cantidad.

Tabla Intermedia Explícita



Usa **ProductSale** como puente.

Victoria Arquitectónica: 'Congela' el **unit_price** en el momento 'T' de la transacción. Permite almacenar atributos propios de la relación: cantidad vendida y total de la línea.

Estado del proyecto: La capa de ventas



ANTES

```
apps/sale/  
├── __init__.py  
├── apps.py  
└── models.py (vacío)
```



DESPUÉS

```
apps/sale/  
├── __init__.py  
├── apps.py  
├── managers.py  [+] NUEVO  
└── models.py    [*] REESCRITO
```

El gestor (**managers.py**) nace hoy, y los modelos encapsularán toda la lógica matemática y de estado.

Preparando el terreno: SaleManager

apps/sale/managers.py

```
class SaleQuerySet(models.QuerySet):  
    pass # register() vendrá en ISS-11  
  
class SaleManager(models.Manager.from_queryset(SaleQuerySet)):  
    pass
```

Nota Arquitectónica

En este ISS el gestor nace vacío. En el **ISS-11** inyectaremos el método `register()` aquí. Aislar la lógica transaccional de creación fuera de las vistas y modelos es una práctica de diseño senior.

Modelo Sale (Cabecera): Relaciones y Finanzas

apps/sale/models.py

```
class Sale(models.Model):  
    # ... other fields  
    client = models.ForeignKey(Client, on_delete=models.RESTRICT, related_name='sales')  
    # ... other fields  
    subtotal = models.DecimalField(max_digits=12, decimal_places=2, default=0.00)  
    tax = models.DecimalField(max_digits=12, decimal_places=2, default=0.00)  
    discounts = models.DecimalField(max_digits=12, decimal_places=2, default=0.00)  
    total = models.DecimalField(max_digits=12, decimal_places=2, default=0.00)  
    # ... other methods
```

Regla de negocio: NUNCA borrar un cliente que tenga ventas asociadas. RESTRICT protege la base de datos.

Tipos precisos. Nunca usar FloatField para dinero.

Modelo Sale (Cabecera): Integridad en Base de Datos

```
constraints = [  
    StoreLabModel.status_constraint('sales_status_valid'),  
    models.CheckConstraint(  
        condition=models.Q(subtotal__gte=0) & models.Q(total__gte=0)...,  
        name='sales_amounts_non_neg',  
    ),  
]
```



CheckConstraint.

La regla vive físicamente en PostgreSQL/MySQL. Incluso si un script de Python falla, la base de datos **rechazará matemáticamente** una venta con totales negativos.

Anatomía de la Anulación: Concurrencia y Atomicidad

```
def void(self):  
    with transaction.atomic():  
        # Lógica de anulación...
```

Atomic Boundary

Lock (select_for_update)

Revert (stock += quantity)

Save (status = INACTIVE)

transaction.atomic(): Todo o nada. Si la reposición de stock del producto 10 falla, las anulaciones de los productos 1 al 9 se revierten (**Rollback**) instantáneamente.

Evitando Condiciones de Carrera (Race Conditions)

```
sale = Sale.objects.select_for_update().get(pk=self.pk)
# ...
product = Product.objects.select_for_update().get(pk=line.product_id)
```

Product

Product Row



User A



User B

`select_for_update()` aplica un bloqueo de fila en la BD. Garantiza que ningún otro proceso paralelo mutará el stock de este producto mientras calculamos la reversión.

Reversión Matemática y Soft Delete

```
product.stock += line.quantity  
product.save(update_fields=['stock', 'updated_at'])  
line.status = RecordStatus.INACTIVE
```

Product

1. Reversión

Devolvemos el **stock** exacto que se vendió en esta línea.

2. Soft Delete

No hacemos `delete()`. Cambiamos el estado a **INACTIVE** para preservar la auditoría.

3. Eficiencia

`update_fields` obliga al ORM a actualizar solo esas columnas, optimizando el ancho de banda y previniendo la sobrescritura accidental de otros campos.

Modelo ProductSale (Detalle): Congelando el Tiempo

```
class ProductSale(models.Model):
    sale = models.ForeignKey(Sale, on_delete=models.RESTRICT)
    product = models.ForeignKey(Product, on_delete=models.RESTRICT)
    quantity = models.PositiveIntegerField(validators=[MinValueValidator(1)])
    unit_price = models.DecimalField(max_digits=10, decimal_places=2)
    line_total = models.DecimalField(max_digits=12, decimal_places=2)

    def save(self, *args, **kwargs):
        # Pre-calculate line_total before saving
        self.line_total = self.quantity * self.unit_price
        super().save(*args, **kwargs)
```

Doble FK

sale y product.
Ambos con
RESTRICT.
Forman la tabla
de intersección.

Campos Congelados

quantity: Validador mínimo de 1.

unit_price: El secreto del historial. Copia exacta del en el milisegundo de la transacción.

line_total: Monto precalculado para evitar uniones costosas (JOINS) en el futuro.

Consistencia Matemática Garantizada

$$[\text{Cantidad}] \times [\text{Precio Unitario}] = [\text{Total de Línea}]$$

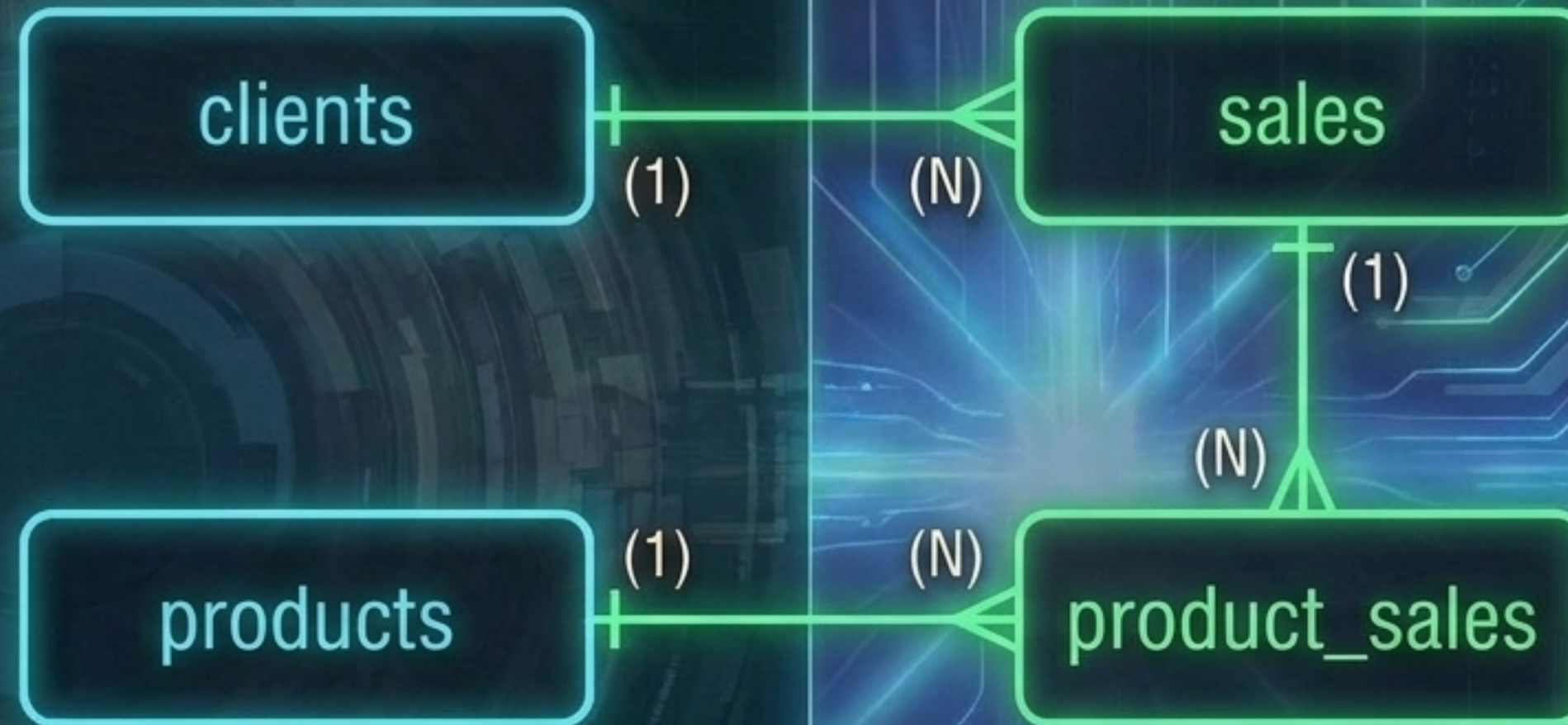


```
def save(self, *args, **kwargs):  
    self.line_total = (self.unit_price * self.quantity).quantize(Decimal('0.01'))  
    super().save(*args, **kwargs)
```

Product

Sobreescribimos `save()` para aislar la regla de negocio. Garantiza que la BD rechace un total falso o inyectado maliciosamente por una vista defectuosa o un script.

Esquema Físico Resultante



La cardinalidad real en la base de datos física.
Un esquema altamente normalizado y protegido transaccionalmente.

Comandos CLI y Migraciones

```
> python manage.py makemigrations sale
Migrations for 'sale':
  apps/sale/migrations/0001_initial.py
  - Create model Sale
  - Create model ProductSale

> python manage.py migrate
Operations to perform:
  Apply all migrations: sale
Running migrations:
  Applying sale.0001_initial... OK
```

Traducimos los planos de Python (DDL ORM)
a estructuras físicas en PostgreSQL/MySQL.

Errores Arquitectónicos Frecuentes



Delegar el historial al catálogo: Usar ManyToManyField y perder el precio al que realmente se vendió el ítem hace meses.



Ignorar concurrencia: Actualizar stock de inventario sin `select_for_update()`, permitiendo condiciones de carrera y 'stock negativo fantasma'.



Destrucción de evidencia: Realizar `delete()` físico en lugar de anulación lógica (`status = INACTIVE`), rompiendo la trazabilidad contable.



Cálculos manuales en vistas: No encapsular `line_total` en el `save()` del modelo, permitiendo inconsistencias matemáticas.

Criterios de Aceptación (AC)

- [✓] AC-06-01: Tablas físicas sales y product_sales creadas correctamente en BD.
- [✓] AC-06-02: Todas las Foreign Keys configuradas con restricción RESTRICT.
- [✓] AC-06-03: line_total se calcula automáticamente dentro del método save.
- [✓] AC-06-04: Ausencia total de ManyToManyField genéricos; se utiliza modelo intermedio.

Evidencias del ISS

EVI-06-01

```
Applying sale.0001_initial... OK
```

Migración aplicada exitosamente en PostgreSQL.

EVI-06-02

```
test_detail_keeps_historical_price ... PASS
```

```
-----  
Ran 1 test in 0.045s
```

```
OK
```

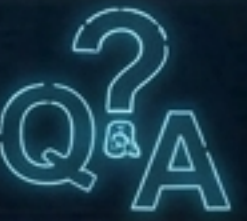
El código demuestra empíricamente que el precio congelado sobrevive a las mutaciones futuras del catálogo.

Verificación de Calidad: GATE

Criterio	Evidencia	Resultado
AC-06-01 (Migración)	EVI-06-01	PASS
AC-06-02 (on_delete)	Modelos	PASS
AC-06-03 (ProductSale.save)	Test de totales	PASS
AC-06-04 (Sin ManyToMany)	sale/models.py	PASS

Compuerta de calidad superada. Arquitectura de ventas lista para endpoints en ISS-11 e ISS-12.

Defensa Oral del Diseño



1. ¿Por qué es estructuralmente peligroso anular una venta (actualizar stock) sin envolver la operación en `select_for_update()`?
2. ¿Qué pasaría físicamente con nuestra base de datos si hubiéramos usado el `ManyToManyField` genérico de Django para el detalle de venta?
3. ¿Por qué delegar el cálculo de `line_total` al método `save()` del modelo en lugar de calcularlo en el controlador/vista?

La maestría no está en escribir código, sino en defender por qué se escribió así.