EXPEDIENTE TÉCNICO DE DB ARQUITECTURA DE DATOS

Proyecto: SIGLO-F (Sistema Integral de Gestión Logística y Fidelización)
Enfoque de Persistencia: Híbrido (JPA para Dominio + SQL Nativo para Rendimiento)
Tecnología: Java 17 + Spring Boot 3 + PostgreSQL 15

---

1. ESTÁNDARES DE DESARROLLO Y PERSISTENCIA

1.1 Estrategia Híbrida de Acceso a Datos

Se define una arquitectura de persistencia dual para balancear mantenibilidad y performance:

A. Capa Transaccional (Hibernate / Spring Data JPA)

- **Uso:** Exclusivo para operaciones **CRUD** (Crear, Leer uno, Actualizar, Borrar) y validaciones de negocio unitarias.
- **Justificación:** Aprovechar el mapeo automático de entidades, manejo de cascadas y consistencia del contexto de persistencia.
- **Ejemplo:** clientRepository.save(client), userRepository.findById(id).

B. Capa Analítica y Masiva (SQL Nativo / JDBC Template)

- **Uso:** Obligatorio para **Reportes**, **Dashboards**, **Cálculos Masivos** (Batch) y consultas con más de 3 JOINs.
- **Implementación:** Uso de anotación @Query(value = "...", nativeQuery = true) o NamedParameterJdbcTemplate para proyecciones DTO (Interfaces) sin mapear a Entidades completas.
- **Justificación:** Evitar el problema "N+1 Selects" y la sobrecarga de memoria de Hibernate al hidratar miles de objetos para solo leer un dato.

1.2 Formato de Respuesta JSON (Estándar)

```
{
  "status": 200,
  "message": "Operación exitosa",
  "data": { ... },
  "timestamp": "2026-01-14T10:00:00"
}
```

2. MÓDULO DE SEGURIDAD (IAM) - (Enfoque JPA)

Este módulo mantendrá un enfoque 100% JPA debido a que la seguridad requiere consistencia estricta de objetos.

- **Entidades:** User, Role.
- **Operaciones:**
  - Login: JPA (Búsqueda por username).
  - Creación de Usuarios: JPA (Para aprovechar validaciones @PrePersist).

3. MÓDULO DE CLIENTES Y SEMÁFORO - (Enfoque Híbrido)

3.1 Modelo de Datos (JPA)

- **Entidad:** Client (Mapeo ORM completo).

3.2 Lógica del Semáforo (Optimización SQL Nativo)

El cálculo diario del estado del cliente (Verde/Gris/Rojo) **NO** se hará trayendo todos los clientes a la memoria de Java (lo cual sería lento e ineficiente).

- **Implementación Técnica:** Se ejecutará una **Query Nativa de Actualización Masiva** programada (@Scheduled).
- **Consulta SQL (Ejemplo Conceptual):**

```
SQL
UPDATE clients
SET status_commercial = CASE
    WHEN last_purchase_date >= NOW() - INTERVAL '15 days' THEN 'FREQUENT'
    WHEN last_purchase_date >= NOW() - INTERVAL '35 days' THEN 'ACTIVE'
    ELSE 'RISK'
END
WHERE status != 'INACTIVE';
```

- **Beneficio:** Procesa 5,000 clientes en milisegundos directamente en el motor de PostgreSQL sin consumir RAM del servidor Java.

4. MÓDULO DE INVENTARIO Y ALMACÉN - (Enfoque JPA)

- **Entidades:** Product, InventoryMovement.
- **Regla:** Debido a la criticidad de la "Ley de Conservación" (el stock no debe perderse), se usará JPA con bloqueo pesimista (@Lock(LockModeType.PESSIMISTIC_WRITE)) durante las transacciones de movimiento para evitar condiciones de carrera.

5. MÓDULO DE RUTAS Y VENTAS (CORE OPERATIVO) - (Enfoque SQL Nativo)

Este es el módulo más crítico y complejo. Aquí el SQL Nativo es protagonista.

5.1 Consultas de Liquidación (Native Query)

Al cerrar una ruta (/routes/close), se requiere cruzar datos de Sales, SaleDetails, Products, Payments y Routes. Hacerlo con JPA generaría demasiadas consultas a la BD.

- **Requerimiento:** Usar **Proyecciones (Interfaces)** en Spring Data para mapear resultados de queries complejas directamente.
- **Ejemplo de Query (Balance de Carga):**

```SQL
SELECT
   p.name as productName,
   SUM(sd.quantity) as totalSold,
   (r.initial_load ->> p.id)::int as initialQty
FROM sales s
JOIN sale_details sd ON s.id = sd.sale_id
JOIN products p ON sd.product_id = p.id
WHERE s.route_id = :routeId
GROUP BY p.name, p.id;
```

5.2 Dashboard de Operaciones (Lectura Rápida)

Para la vista móvil del repartidor ("Mis Ventas de Hoy"), se usará SQL nativo para obtener solo las columnas necesarias (DTOs planos) sin traer las relaciones de hibernate (lazy loading) que ralentizan la app móvil.

6. MÓDULO DE FIDELIZACIÓN (QR) - (Enfoque Híbrido)

- **Asignación de Puntos:**
  - Se usará un **Trigger de Base de Datos** o una operación SQL Nativa insertada post-venta para sumar puntos atómicamente.
  - *Razón:* Evitar que un error en la capa de aplicación Java deje una venta registrada pero sin puntos asignados.

9. REQUERIMIENTOS DE SEGURIDAD (Actualizado)

1. **Prevención de SQL Injection:**
   - Al usar SQL Nativo, está **prohibido concatenar Strings** ("SELECT * FROM t WHERE id=" + id).
   - **Obligatorio:** Uso de parámetros nombrados (:userId, :date) gestionados por NativeQuery o PreparedStatement.
2. **Transaccionalidad Mixta:**
   - Uso de @Transactional en los Servicios que mezclan llamadas a Repositorios JPA y ejecuciones de JDBC Template para asegurar que, si falla el SQL nativo, el JPA haga rollback (y viceversa).