Enunciado Corte 2: Sistema de Consulta Unificada "ServiCiudad Cali"

Contexto del Problema

"ServiCiudad Cali" enfrenta una crisis de servicio al cliente debido a sus sistemas tecnológicos obsoletos y aislados. [1][1][1][1] Una de las quejas más comunes de los ciudadanos es tener que contactar a tres canales diferentes para conocer sus saldos de Energía, Acueducto y Telecomunicaciones. [2] De hecho, el 60% de las llamadas al contact center son para consultas de saldo, un proceso que podría ser automatizado. [3]

Tu equipo ha sido contratado para desarrollar un primer prototipo funcional que demuestre cómo la tecnología puede resolver este problema. Construirás una aplicación **monolítica** con Spring Boot que centralizará la consulta de los servicios más críticos: Energía y Acueducto.

Objetivo del Ejercicio

Desarrollar una API RESTful que ofrezca un punto de acceso único (endpoint) para que un cliente, usando su número de identificación, pueda obtener un resumen consolidado de su deuda de los servicios de **Energía** y **Acueducto**.

Requerimientos Funcionales

La aplicación debe simular la interacción con dos de los sistemas legados de ServiCiudad:

1. **Sistema de Energía (Mainframe IBM Z):**
    - Este sistema no tiene API. [4][4][4][4] Su única salida de datos es un **archivo plano (`consumos_energia.txt`)** que se genera diariamente.
    - Deberás **leer y procesar este archivo** para obtener la información de facturación de energía. El archivo tiene un formato de ancho fijo como el siguiente:
    - Plaintext

```
// id_cliente(10), periodo(6), consumo_kwh(8), valor_pagar(12)
0001234567202510000015000001800000.50
0009876543202510000022500002700000.75
0001122334202510000008000000960000.00
```

    -
    -
2. **Sistema de Acueducto (Base de Datos Oracle):**

- Este sistema utiliza una base de datos Oracle.[5] Para este ejercicio, simularemos esta fuente de datos utilizando una base de datos **PostgreSQL**.
- Deberás crear una tabla `facturas_acueducto` y poblarla con datos de ejemplo. La aplicación deberá conectarse a esta base de datos para obtener la deuda del servicio de agua.
- Estructura de la tabla `facturas_acueducto`:
  - `id` (serial, primary key)
  - `id_cliente` (varchar)
  - `periodo` (varchar)
  - `consumo_m3` (integer)
  - `valor_pagar` (decimal)

3. **API de Consulta Unificada:**
   - Debes exponer un único endpoint: `GET /api/v1/clientes/{clienteId}/deuda-consolidada`
   - Al recibir una solicitud, la aplicación deberá:
     a. Buscar la deuda de energía para el clienteId en el archivo plano.
     b. Buscar la deuda de acueducto para el clienteId en la base de datos PostgreSQL.
     c. Combinar la información y devolver una respuesta JSON consolidada.
   - **Ejemplo de respuesta JSON:**
   - JSON

```
{
  "clienteId": "0001234567",
  "nombreCliente": "Juan Pérez", // Puedes usar un valor fijo o mock
  "fechaConsulta": "2025-10-26T10:00:00Z",
  "resumenDeuda": {
    "energia": {
      "periodo": "202510",
      "consumo": "150 kWh",
      "valorPagar": 180000.50
    },
    "acueducto": {
      "periodo": "202510",
      "consumo": "15 m³",
      "valorPagar": 95000.00
    }
  },
  "totalAPagar": 275000.50
}
```

   -
   -

Requerimientos Técnicos y Patrones a Implementar

Debes implementar y justificar el uso de **al menos 5 patrones de diseño**. Dos de ellos pueden ser patrones que Spring Boot provee de forma nativa, pero debes explicar su función y beneficio en tu solución.

1. **Patrón Adapter:** El archivo plano del mainframe es un sistema legado con una interfaz (formato de texto de ancho fijo) incompatible con tu lógica de negocio, que trabaja con objetos Java. Debes crear una clase `AdaptadorArchivoEnergia` que lea el archivo y convierta (adapte) sus datos a una lista de objetos `FacturaEnergia` que el resto de la aplicación pueda entender.[6]
2. **Patrón Builder:** El objeto JSON de respuesta (`DeudaConsolidadaDTO`) es complejo y se construye a partir de múltiples fuentes de datos (energía, acueducto, datos del cliente). Utiliza el patrón Builder para construir este objeto de respuesta de forma legible, paso a paso, y mantener el proceso de construcción separado de la representación final.[7]
3. **Patrón Data Transfer Object (DTO):** La respuesta de tu API no debe exponer directamente las entidades de la base de datos o los objetos internos. Crea clases DTO específicas (ej. `DeudaConsolidadaDTO`, `DetalleServicioDTO`) para modelar la estructura de la respuesta JSON que el cliente espera.
4. **Patrón Repository (Provisto por Spring):** Para interactuar con la base de datos PostgreSQL, utilizarás Spring Data JPA. Debes crear una interfaz que extienda `JpaRepository`. En tu informe, **justifica por qué este patrón abstrae la complejidad del acceso a datos** y facilita la implementación de operaciones CRUD sin código repetitivo (boilerplate).
5. **Inversión de Control / Inyección de Dependencias (Provisto por Spring):** El framework Spring Boot gestiona el ciclo de vida de los componentes (`@Service`, `@Repository`, `@RestController`). En tu informe, **explica cómo utilizaste la inyección de dependencias** (ej. inyectando el servicio en el controlador) y por qué este patrón es fundamental para lograr un bajo acoplamiento y alta cohesión en tu aplicación.

Pila Tecnológica Sugerida

- **Lenguaje:** Java 17+
- **Framework:** Spring Boot 3.x
- **Base de Datos:** PostgreSQL
- **Gestión de Dependencias:** Maven o Gradle
- **Librerías Adicionales:** Spring Data JPA, Spring Web, Lombok (opcional).

Entregables

1. **Código Fuente:** Repositorio en GitHub con el proyecto completo.

2. **Archivo `README.md`:** Instrucciones claras sobre cómo clonar, configurar la base de datos y ejecutar el proyecto.
3. **Informe (`INFORME.md`):** Un documento breve dentro del repositorio donde expliques:
    - La arquitectura general de tu monolito.
    - La justificación de cómo y por qué aplicaste cada uno de los 5 patrones de diseño solicitados.
4. **Colección de Postman:** Un archivo de exportación de Postman con las solicitudes para probar el endpoint.