Memoria Técnica Descriptiva - HybridLogisticsHub

Sistema de Gestión Logística con Arquitectura Híbrida PostgreSQL + MongoDB

Jhordan Huamaní Huamaní[1], Jorge Ortiz Castañeda[1], Miguel Flores León[1]
[1]Departamento de Ingeniería de Software, Universidad La Salle de Arequipa, Perú
{jhordanhh, jortizc, mfloresl}@ulasalle.edu.pe

***Resumen*—Este documento presenta la memoria técnica del sistema HybridLogisticsHub. Se detalla la arquitectura híbrida implementada, combinando PostgreSQL para datos transaccionales y MongoDB para datos geoespaciales, así como el modelado de datos, la API REST y las estrategias de despliegue.**

***Index Terms*—Arquitectura Híbrida, PostgreSQL, MongoDB, FastAPI, Geoespacial, Docker.**

I. Introducción

I-A. Objetivo del Sistema

HybridLogisticsHub es un sistema de gestión logística diseñado para demostrar la implementación de una **arquitectura de bases de datos híbrida**, combinando:

- **PostgreSQL**: Base de datos relacional para datos transaccionales.
- **MongoDB**: Base de datos NoSQL para datos geoespaciales y tracking en tiempo real.

I-B. Alcance

El sistema abarca:

- Gestión de clientes y órdenes (CRUD completo).
- Tracking de ubicación en tiempo real.
- Consultas geoespaciales (búsqueda por cercanía y zona).
- Visualización interactiva con mapas.
- Simulación de entregas con rutas reales.

II. Arquitectura General

II-A. Diagrama de Arquitectura

La arquitectura del sistema, mostrada en la Fig. 1, separa claramente el frontend, el backend y la capa de persistencia híbrida.

II-B. Patrón de Arquitectura

El sistema implementa una arquitectura en capas detallada en la Tabla I.

Cuadro I
Capas del Sistema

Capa	Responsabilidad	Componentes
Presentación	Interfaz de usuario	HTML, Leaflet.js
API	Endpoints	FastAPI Routes
Servicios	Lógica de negocio	Services Python
Modelos	Validación de datos	Pydantic Schemas
Datos	Persistencia	PostgreSQL, MongoDB

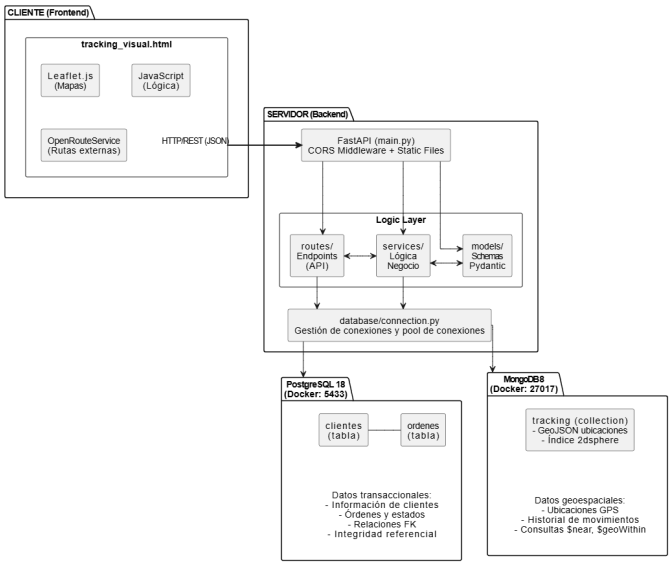

Figura 1. Diagrama de Arquitectura

II-C. Justificación de la Arquitectura Híbrida

- **PostgreSQL:** Ideal para datos estructurados, integridad referencial y transacciones ACID (Clientes, Órdenes).
- **MongoDB:** Ideal para datos semi-estructurados de alta velocidad de escritura y consultas geoespaciales complejas (Tracking GPS).

III. Tecnologías Utilizadas

III-A. Stack Tecnológico

El backend está construido sobre Python 3.10+, utilizando **FastAPI** como framework principal debido a su rendimiento y soporte nativo asíncrono. Para la persistencia se utilizan los drivers `psycopg2` (PostgreSQL) y `pymongo` (MongoDB).

La infraestructura se gestiona mediante **Docker** y **Docker Compose**, orquestando los contenedores de base de datos y aplicación.

IV. Modelado de Datos

IV-A. Modelo Relacional (PostgreSQL)

Se utilizan dos tablas principales con relación 1:N, como se aprecia en la Fig. 2.

Los estados válidos de una orden son: Pendiente, En Proceso, En Tránsito, Entregado y Cancelado.

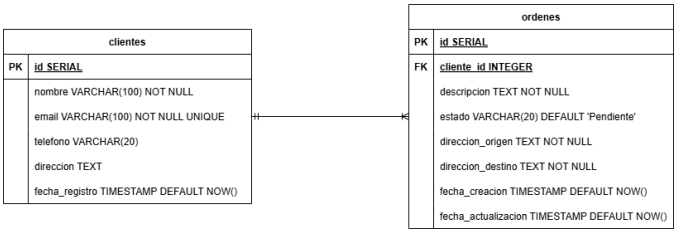

Figura 2. Modelo Relacional

```
ESTADOS_ORDEN_VALIDOS = [
    "Pendiente",    # Orden creada, esperando procesamiento
    "En Proceso",   # Orden siendo preparada
    "En Tr nsito",  # Orden en camino
    "Entregado",    # Orden entregada exitosamente
    "Cancelado"     # Orden cancelada
]
```

IV-B. Modelo de Documentos (MongoDB)

La colección `tracking` almacena puntos geoespaciales.

```
{
  "_id": ObjectId("..."),
  "orden_id": 1,
  "ubicacion": {
    "type": "Point",
    "coordinates": [-71.537, -16.409]
  },
  "timestamp": ISODate("2025-11-30T10:30:00Z"),
  "velocidad_kmh": 45.5
}
```

Listing 1. Ejemplo de Documento Tracking

Se utiliza un índice geoespacial `2dsphere` en el campo `ubicacion` para optimizar las consultas de proximidad.

IV-C. Relación entre Bases de Datos

La relación es lógica y se maneja a nivel de aplicación (Servicios). El campo `orden_id` en MongoDB referencia a la PK `id` de la tabla `ordenes` en PostgreSQL (ver Fig. 3).

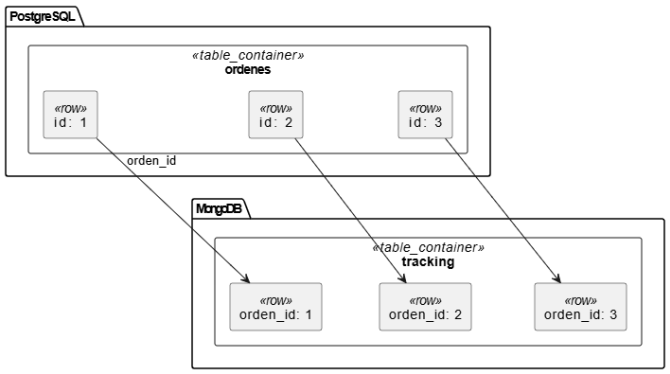

Figura 3. Relación entre Bases de Datos

V. Modelado de Procesos

V-A. Flujo: Crear Orden

El proceso de creación de orden, detallado en la Fig. 4, implica la validación de datos mediante Pydantic y la inserción transaccional en PostgreSQL.

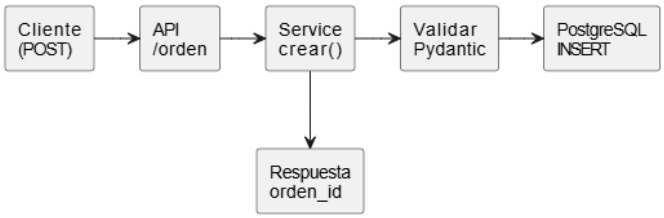

Figura 4. Flujo: Crear Orden

V-B. Flujo: Registrar Tracking y Consultas

El registro de tracking ingresa directamente a MongoDB, tal como se muestra en la Fig. 5.

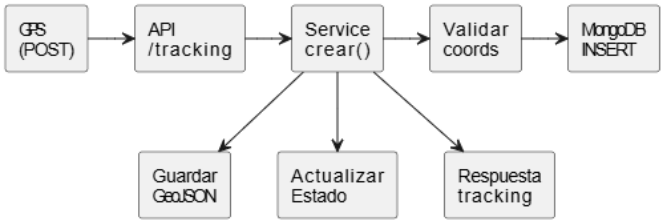

Figura 5. Flujo: Registrar Tracking

Para las consultas geoespaciales (ej. "Órdenes cercanas"), el sistema realiza una operación federada ilustrada en la Fig. 6:

1. Consulta MongoDB usando `$near` para obtener IDs.
2. Consulta PostgreSQL para obtener detalles de esas órdenes.
3. Fusiona los resultados en la capa de servicios.

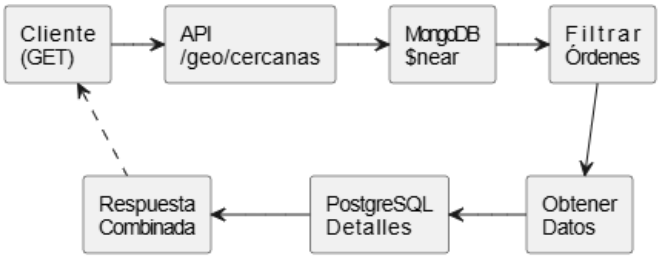

Figura 6. Flujo: Consulta Geoespacial

Finalmente, el flujo de simulación de entrega en el frontend se visualiza en la Fig. 7, conectando el cliente con la API de rutas.

VI. API REST - Endpoints

La API expone los recursos listados en la Tabla II.

VII. Validaciones y Reglas de Negocio

VII-A. Validaciones Pydantic

Se utilizan esquemas estrictos para validar entradas. Por ejemplo, las coordenadas deben estar dentro de rangos geográficos válidos y los estados de las órdenes deben pertenecer a la lista predefinida.

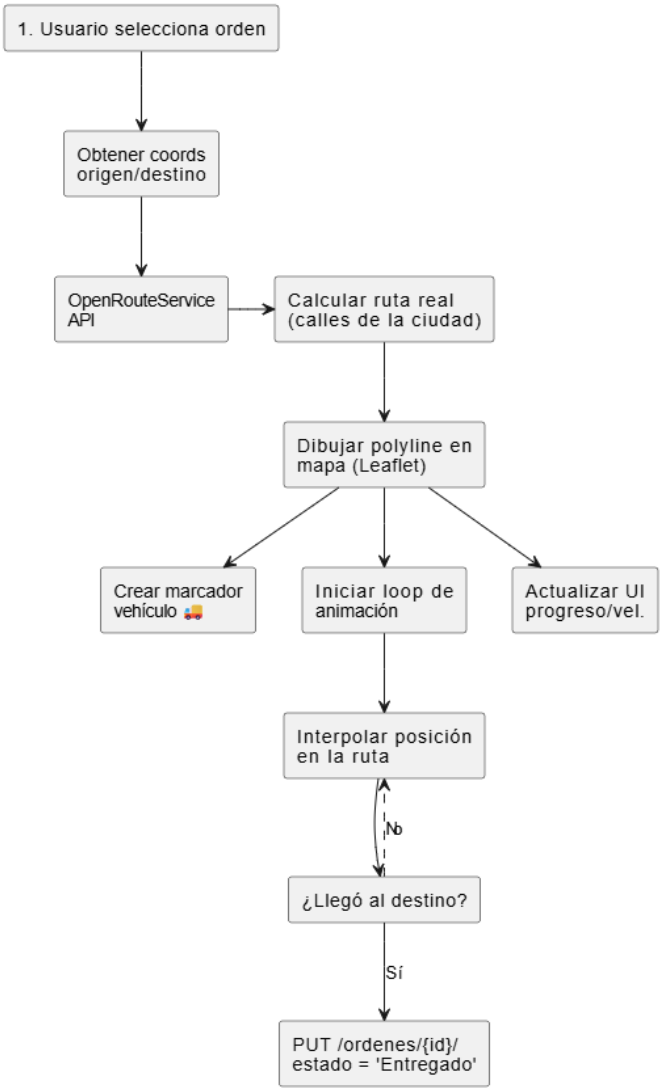

Figura 7. Flujo: Simulación de Entrega (Frontend)

Cuadro II
RESUMEN DE ENDPOINTS PRINCIPALES

Método	Endpoint	Base de Datos
GET	/clientes	PostgreSQL
POST	/ordenes	PostgreSQL
GET	/ordenes/{id}	PostgreSQL
PUT	/ordenes/{id}/estado	PostgreSQL
POST	/tracking	MongoDB
GET	/tracking/{id}/historial	MongoDB
GET	/geo/cercanas	Híbrida

VII-B. Transiciones de Estado

Se implementa una máquina de estados simple para evitar transiciones ilógicas (ej. no se puede pasar de .[En]tregado.[a] "Pendiente").

VIII. SEGURIDAD

- **CORS:** Configurado para permitir peticiones desde el frontend.
- **Variables de Entorno:** Las credenciales de base de datos no se queman en código, sino que se inyectan mediante `os.getenv`.
- **Inyección SQL:** Prevenida mediante el uso de ORM y query builders parametrizados.

IX. INSTALACIÓN Y DESPLIEGUE

IX-A. Requisitos

Se requiere Docker y Python 3.10+.

IX-B. Instalación Local

```
# Clonar y levantar servicios
git clone https://github.com/JorLOrT/Repo.git
docker-compose up -d

# Inicializar DB y ejecutar
pip install -r requirements.txt
python init_db.py
python -m uvicorn main:app --reload
```

Listing 2. Comandos de Instalación

IX-C. Despliegue con Docker Compose

El archivo `docker-compose.yml` define tres servicios: `postgres`, `mongodb` y la `api`. Esto asegura que el entorno sea reproducible en cualquier servidor.

X. CONCLUSIONES

X-A. Ventajas del Sistema

La arquitectura híbrida permite aprovechar lo mejor de dos mundos: la robustez transaccional de SQL y la flexibilidad geoespacial de NoSQL. FastAPI facilita un desarrollo rápido y documentación automática.

X-B. Mejoras Futuras

- Implementación de autenticación JWT.
- Uso de WebSockets para tracking en tiempo real (push vs pull).
- Caché con Redis para consultas frecuentes.

XI. ANEXOS

XI-A. Estructura de Archivos

```
HybridLogisticsHub/
 main.py             # Entry point
 database/           # Conexiones
 models/             # Schemas
 routes/             # API Endpoints
 services/           # Lógica
 docker-compose.yml  # Infraestructura
```