Manual Técnico - Arquitectura y Base de Datos - Gravio

1. Arquitectura del Sistema

Gravio sigue los principios de **Clean Architecture** (Arquitectura Limpia) adaptada para una aplicación de escritorio con Electron. El objetivo es desacoplar la lógica de negocio de la interfaz de usuario y de la infraestructura externa (hardware, base de datos).

1.1. Capas del Sistema

1. **Domain (Dominio)**: `src/domain/`
   - Núcleo del sistema. Contiene las reglas de negocio puras.
   - **Entities**: Objetos de negocio como `Registro`, `Vehiculo`, `Empresa`, `Operador`.
   - **Repositories (Interfaces)**: Contratos que definen cómo se accede a los datos (`IRegistroRepository`, `IVehiculoRepository`).
   - *Dependencias*: Cero. No conoce React, ni Electron, ni SQL.
2. **Application (Aplicación)**: `src/application/`
   - Casos de uso y orquestación.
   - **Services**: `PesajeService` (lógica de pesaje), `SyncService` (coordinación de sincronización).
   - **DIContainer**: Inyección de dependencias para conectar las implementaciones concretas con las interfaces.
3. **Infrastructure (Infraestructura)**: `src/infrastructure/`
   - Implementaciones concretas de los repositorios y adaptadores.
   - **Database**: `SQLiteRegistroRepository` (Local), `SupabaseRegistroRepository` (Nube).
   - **Hardware**: `MettlerToledoScale` (Lector serial), `PrinterService` (ESC/POS).
4. **Presentation (UI)**: `src/components/`
   - Interfaz de usuario construida con **React 18**.
   - Utiliza **TailwindCSS** para estilos y **Lucide Icons**.
   - Se comunica con la capa de Aplicación a través del `DIContainer` o Hooks personalizados.
5. **Main Process (Electron)**: `electron/`
   - `main.ts`: Punto de entrada. Gestiona ventanas y ciclo de vida.
   - `preload.ts`: Puente seguro (ContextBridge) entre el proceso Main (Node.js) y el Renderer (React). Expone APIs seguras como `window.electron.serialPort`.

---

2. Base de Datos y Esquema

El sistema utiliza un enfoque híbrido:

- **SQLite (`better-sqlite3`)**: Base de datos principal en el cliente. Garantiza funcionamiento offline.
- **Supabase (PostgreSQL)**: Base de datos en la nube para centralización y respaldo.

2.1. Tablas Principales (Schema)

`registros`

Tabla transaccional principal. Almacena cada evento de pesaje.

- `id` (UUID): Identificador único global.
- `folio` (TEXT): Identificador legible humano (ej. `GRAV-00123`).
- `tipo_pesaje` (TEXT): `'entrada'`, `'salida'`, `'completo'`.
- `peso_entrada` (REAL): Peso en kg al ingresar.
- `peso_salida` (REAL): Tara del vehículo al salir (nullable).
- `peso_neto` (REAL): Calculado (`entrada - salida`).
- `placa_vehiculo` (TEXT): Placa del camión.
- `clave_empresa`, `clave_operador`, `clave_ruta`: Referencias a catálogos.
- `sincronizado` (BOOLEAN): Bandera de control para el motor de sync (`0` = pendiente, `1` = synced).

`vehiculos`

Catálogo de unidades autorizadas.

- `id` (UUID)
- `placas` (TEXT)
- `no_economico` (TEXT): Identificador interno de la empresa.
- `tara_actual` (REAL): Último peso conocido vacío (para referencia).

`operadores`

Catálogo de choferes.

- `id` (UUID)
- `operador` (TEXT): Nombre completo.
- `clave_operador` (INTEGER): ID numérico legado.

`folio_sequences`

Control de folios offline.

- `clave_empresa` (INTEGER)
- `ultimo_numero` (INTEGER): Contador incremental para generar folios sin conexión.
- `prefijo_empresa` (TEXT): Ej. "GRAV".

`sync_queue`

Cola de prioridad para la sincronización.

- `id` (INTEGER PK)
- `entity_type` (TEXT): 'registro', 'vehiculo', etc.

- `entity_id` (TEXT): ID del registro afectado.
- `operation` (TEXT): 'INSERT', 'UPDATE'.
- `created_at` (DATETIME).

---

3. Motor de Sincronización (Sync Engine)

El `SyncService` (`src/application/services/SyncService.ts`) y `sync.ts` (`src/lib/sync.ts`) orquestan el flujo de datos.

Flujo de Sincronización (Subida / Upstream)

1. El usuario guarda un registro. Se escribe en **SQLite** y se marca `sincronizado = 0` .
2. Un *watcher* o el temporizador (cada 5 min) detecta conexión a internet.
3. Se leen todos los registros con `sincronizado = 0` .
4. Se envían uno a uno (o en batch) a **Supabase** usando `SupabaseRegistroRepository` .
5. Si Supabase confirma (HTTP 200/201), se actualiza el registro local a `sincronizado = 1` .

Flujo de Descarga (Downstream)

1. Al iniciar la aplicación, se consultan los catálogos (`Empresas` , `Vehiculos` , `Rutas`) en Supabase.
2. Se comparan timestamps (`updated_at`).
3. Si hay cambios en la nube, se descargan y actualizan en la SQLite local usando `INSERT OR REPLACE` .

---

4. Integración de Hardware

Comunicación Serial (Báscula)

- **Librería**: `serialport` (Node.js).
- **Implementación**: El proceso `Main` de Electron abre el puerto.
- **Eventos**:
  - `data` : Recibe bytes crudos.
  - El parser busca el patrón de trama Mettler Toledo.
  - Se envía un evento IPC `serial-data` al Renderer (React UI).

Impresión Térmica

- **Protocolo**: ESC/POS (Raw bytes).
- **Implementación**: Se construye un buffer de comandos (texto, saltos de línea, corte de papel) y se escribe directamente al puerto USB/Serial de la impresora o se usa el driver de Windows vía `electron-printer` .

---

5. Scripts de Desarrollo

Ubicados en la carpeta `scripts/` :

- `scale_simulator.py` : Script en Python que simula una báscula física. Crea un puerto COM virtual y envía tramas de peso aleatorias. Útil para probar sin hardware.
- `convert_manuals_to_pdf.cjs` : Herramienta interna para generar esta documentación.

Stack Tecnológico:

- Electron 28+
- React 18
- TypeScript 5
- Vite 5
- Better-SQLite3
- Supabase JS Client