Decisiones técnicas – Prueba Concesionaria

1. Objetivo y alcance

La consigna original pedía:

- Una app en React que:
    - Liste vehículos (marca, modelo, precio, imagen).
    - Permite filtrar por marca y rango de precio (con extras opcionales como año o combustible).
    - Muestre el detalle de un vehículo en una ruta o pop-up.
- Una base de datos MySQL con:
    - Tablas para vehículos, clientes, ventas y formas de pago.
    - Datos de prueba.
    - Algunas consultas SQL específicas.
- Como extra, se sugería mostrar en la app de React los datos reales desde la base de datos a través de un backend.

En lugar de resolverlo con mocks simples, decidí construir una **mini aplicación full stack** un poco más cercana a un sistema real de concesionaria:

- **Frontend** en React + TypeScript, con Material UI y Vite.
- **Backend** en Node.js + Express, consumiendo una base MySQL real.
- **Infraestructura local** montada completamente en **Docker** (DB, phpMyAdmin, backend y frontend).

- Implementación de **módulo de ventas**, **formas de pago**, **gestión de imágenes de vehículos**, **filtros más completos** y **flujo de compra**.

---

2. Arquitectura general del sistema

2.1. Stack principal

- **Frontend**:
    - React + Vite.
    - TypeScript para tipado fuerte.
    - Material UI (MUI) para componentes de UI.
- **Backend**:
    - Node.js (LTS 20) + Express.
    - Organización en capas: rutas → controladores → servicios → acceso a datos.
- **Base de datos**:
    - MySQL 8.
    - Uso de **stored procedures** para encapsular lógica crítica de negocio (altas, bajas lógicas, ventas, etc.).
- **Infraestructura local**:
    - `docker-compose` con servicios:
        - `db` (MySQL).
        - `phpmyadmin` para inspeccionar la base.
        - `backend` (API Node/Express).
        - `frontend` (app React).

2.2. Comunicación front–back

- El frontend se comunica con el backend mediante **endpoints REST**:
    - `/api/vehiculos` para listar, filtrar, crear, editar y cambiar estado.
    - `/api/ventas` para registrar ventas y listar ventas.
    - `/api/pagos` o integrado dentro de ventas según el flujo.
    - `/api/auth` / `/api/usuarios` para autenticación y gestión de usuarios.
- El backend traduce estos requests a llamadas a:
    - Consultas SQL directas para listados simples.
    - Stored procedures para operaciones con lógica (ventas, alta/edición de vehículos, manejo de imágenes, etc.).

---

3. Decisiones de Frontend

3.1. Elección de React + Vite + TypeScript

- **React** cumple con la consigna y es una herramienta estándar del mercado.
- **Vite**:
    - Arranque muy rápido de entorno de desarrollo.
    - Build más eficiente que CRA.
- **TypeScript**:
    - Mejora la robustez y el auto-completado.
    - Facilita el contrato de datos entre frontend y backend (por ejemplo, interfaces `Vehicle`, `Sale`, etc.).

3.2. Uso de Material UI (MUI)

Tomé la decisión de usar **Material UI** porque:

- Permite construir interfaces modernas y responsivas rápidamente.
- Ofrece componentes de alto nivel que se ajustan a un diseño limpio (Cards, Dialogs, Tabs, AppBar, Grid, etc.).
- Puede definir un **theme** global (tipografía, colores, radio de bordes) para que toda la app se vea consistente.

3.3. Estructura del frontend

Organización de código en:

- `/components`
    - `/vehicles`
        - `VehicleList.tsx`: listado general con filtros, paginación y layout responsivo.
        - `VehicleCard.tsx`: tarjeta de vehículo con imagen principal, marca, modelo, precio y estado.
        - `VehicleDetailDialog.tsx`: modal con información detallada (más fotos, descripción, características técnicas).
        - `VehicleCreateDialog` / `VehicleEditDialog`: formularios para alta/edición de vehículos (para rol admin-vendedor).
    - `/sales`
        - `SalesDashboard.tsx`: vista para visualizar ventas, filtrar por fecha, ver estados, etc.
        - `SaleCreateDialog.tsx`: flujo de registro de venta con selección de forma de pago y cuotas
    - /mySales:
        - MySalesTable.tsx: Tabla que lista las compras del usuario: fecha, vehículo, estado de la venta, monto total y acceso al detalle de cuotas.

        - MyMovementsTable.tsx: Tabla de movimientos personales del usuario: fecha, tipo (entrada/cuota), método de pago y monto. Se usa para listar pagos, cuotas canceladas y otros movimientos asociados a sus propias compras.

- `/hooks`
    - `useVehicles.ts`: hook para centralizar lógica de fetching, filtros, paginación y operaciones de vehículos.
    - `useSales.ts`: hook para manejar ventas, rangos de fecha, etc.
    - `useAuth.ts`: hook para autenticación (manejo de token, usuario logueado, etc.).
    - useMySales.tsx :Hook flujo de “mis compras”: manejar filtros por fecha, solicitar ventas propias, movimientos propios y coordinar llamadas al servicio. (Basado en la misma estructura de useSales).
- `/services`
    - `vehicleService.ts`: acceso a la API de vehículos.
    - `salesService.ts`: acceso a la API de ventas/pagos.
    - `authService.ts` / `usersService.ts`: para login y usuarios, según tu implementación final.

Esta estructura permite:

- Reutilizar lógica (por ejemplo, `useVehicles` se usa tanto en la vista pública de catálogo como en vistas de administración).
- Mantener el código organizado por dominio funcional (vehicles, sales, auth).

3.4. Listado de vehículos y filtros.

Aunque la consigna pedía solo marca y rango de precio, amplié el sistema de filtros:

- **Filtros implementados**:

    - Marca.
    - Rango de precio.
    - Combustible (Nafta, Diesel, etc.).
    - Tipo de vehículo (auto, camioneta, moto, etc.).
    - Transmisión (manual, automática).
    - Tracción (4x2, 4x4, etc.).
    - Color.
    - Estado (EN_STOCK, EN_EVALUACION, VENDIDO).
    - Km, año, origen, etc.
- Lógica:
    - Los filtros se manejan mediante `useVehicleFilters`.
    - Al cambiar un filtro se reinicia la página a 1 y se llama al endpoint con query params.
    - El backend aplica estos filtros directamente en SQL para que la paginación sea real (no solo en memoria).

3.5. Detalle del vehículo (pop-up) y flujo de compra

En vez de hacer una ruta aparte, elegí un **modal (Dialog)** para:

- Mantener al usuario en el contexto del listado.
- Mostrar más información sin perder el scroll/posición de la grilla.

El modal incluye:

- Imagen principal y, si existen, galería de imágenes adicionales.
- Marca, modelo, precio, año, km y resto de atributos técnicos.

- Botón de **“Comprar”**.

Decisión clave del flujo de compra:

- El botón **“Comprar”** aparece en el detalle del vehículo.
- Al hacer clic:
    - Si el usuario **no está logueado**, se redirige al flujo de login/registro.
    - Si el usuario está logueado:
        - Se abre el **SaleCreateDialog** y se pasa el `id_vehiculo` por props (`defaultVehicleId`) para no tener que volver a seleccionar el auto.
        - Esto asegura que la venta siempre esté atada al vehículo que el usuario estaba viendo.

3.6. Responsividad y UX

- Uso intensivo de **`Grid`** y **`Stack`** de MUI para manejar diferentes breakpoints (mobile, tablet, desktop).
- Ajuste de cantidad de columnas de cards según viewport.

---

4. Decisiones de Backend (Node + Express)

4.1. Organización del backend

Estructura típica:

- `/routes`: define las rutas REST (ej. `vehicles.routes.js`, `sales.routes.js`, `auth.routes.js`).
- `/controllers`: recibe el request, valida parámetros y delega en servicios o stored procedures.

- `/services` : abstrae la lógica de negocio/red SQL.
- `/config/db.js`: pool de conexión MySQL.

Ejemplo de flujo:

1. El frontend llama a `GET /api/vehiculos?marca=...&minPrecio=...`.
2. La ruta delega en `vehiclesController.getAll`.
3. El controlador arma los parámetros y llama al SP o query correspondiente.
4. Devuelve al frontend un JSON ya adaptado al formato que consume el hook `useVehicles`.

4.2. Razón para usar MySQL + stored procedures

- MySQL cumple con la consigna y es un motor muy usado en entornos productivos.
- Se utilizaron **stored procedures** para:
    - Registrar ventas (validando stock, estado del vehículo, etc.).
    - Actualizar el estado del vehículo a “VENDIDO” o “EN_EVALUACION”.
    - Manejar la lógica de imágenes (por ejemplo, garantizar que solo haya una imagen de perfil).
    - Centralizar reglas de negocio, evitando duplicación de lógica entre backend y base.

Esto hace que:

- Las operaciones críticas sean más atómicas.
- Sea más fácil mantener la integridad de los datos.

---

5. Modelo de datos y decisiones de base de datos (MySQL)

5.1. Tablas principales

A partir de la consigna de “stock de vehículos, clientes, ventas y formas de pago” expandí el modelo con tablas adicionales para reflejar mejor una concesionaria real:

- **vehiculos**
    - `id_vehiculo` (PK)
    - `id_marca` (FK a `marcas`)
    - `modelo`
    - `id_combustible` (FK a `combustibles`)
    - `id_color` (FK a `colores`)
    - `id_tipo_vehiculo`
    - `id_transmision`
    - `id_traccion`
    - `anio`
    - `km`
    - `precio`
    - `estado` (EN_STOCK, EN_EVALUACION, VENDIDO, etc.)
    - `origen`
    - `es_usado` (0/1)
- **marcas**, **combustibles**, **colores**, **tipos_vehiculo**, **transmisiones**, **tracciones**

    - Tablas de catálogo para normalizar datos y permitir filtros.

- **clientes** / **usuarios**
    - Información básica de clientes (nombre, email, teléfono, etc.).
    - En algunos casos se reutiliza como usuario del sistema (rol admin, vendedor, cliente).
- **ventas**
    - `id_venta` (PK)
    - `id_vehiculo` (FK a `vehiculos`)
    - `id_cliente` (FK a `clientes`)
    - `fecha_venta`
    - `monto_total`
    - `id_metodo_pago`
    - Campos adicionales para cuotas u otras condiciones.
- **metodos_pago**
    - `id_metodo_pago`
    - `nombre` (Efectivo, Transferencia, etc.)
- **pagos/cuotas** (si lo separaste)
    - Permite modelar casos de pago en varias cuotas.

5.2. Relaciones y claves foráneas

- Se definieron **FKs explícitas** para:
    - Asegurar que una venta siempre esté ligada a un vehículo y a un cliente válido.

    - Que los valores de marca, combustible, color, etc. provengan de catálogos válidos.

- Se usaron `ON UPDATE/DELETE` según convenía:
    - En general, se evitó el borrado físico de vehículos; se usa un campo de estado (baja lógica) para no perder historial de ventas.

5.3. Datos de prueba y queries pedidas

- Se cargaron al menos **20 registros combinados** entre clientes, autos y ventas, cumpliendo la consigna.
- Se implementaron las 3 consultas requeridas:
    - Todas las ventas de un mismo cliente.
    - Todos los vehículos con año > 2020.
    - Todas las ventas con forma de pago “Efectivo”.
- Además de eso, se añadieron consultas y/o vistas adicionales para:
    - Soportar filtrado por múltiples atributos.
    - Obtener el vehículo con su **imagen de perfil** juntando `vehiculos` con `vehiculos_imagenes`.

---

6. Gestión de imágenes de vehículos (con ImageKit)

Se decidió **no almacenar imágenes en la base de datos ni en el servidor**, sino usar un servicio externo de imágenes: **ImageKit**.

6.1. Modelo en base de datos

- Tabla `vehiculos_imagenes`:
    - `id_imagen` (PK).

- `id_vehiculo` (FK a `vehiculos`).
- `url_imagen` (**URL completa generada por ImageKit**).
- `img_perfil` (booleano) → indica cuál es la imagen principal del vehículo.
- `orden` (número entero) → para definir el orden en la galería.

La **base de datos solo guarda la URL** (string) que devuelve ImageKit.
Esto facilita:

- Independencia entre la app y el lugar físico donde se alojan las imágenes.
- Poder cambiar de proveedor en el futuro sin cambiar el modelo de datos (solo el código que genera URLs).

6.2. Flujo de subida de imágenes

1. En el **frontend**, al crear o editar un vehículo, el usuario puede:
   - Agregar nuevas imágenes.
   - Cambiar la imagen de perfil.

2.Cuando el usuario selecciona un archivo de imagen:

- El frontend lo sube a **ImageKit** usando el SDK / endpoint correspondiente.
- ImageKit devuelve una **URL optimizada y lista para usar**.
- El frontend envía esa URL al backend en un payload del estilo:

```
{
  "id_vehiculo": 123,
  "url_imagen":
"https://ik.imagekit.io/xxxx/vehiculos/auto123-1.jpg",
  "img_perfil": true,
  "orden": 1
}
```

2. El backend guarda esa información en la tabla `vehiculos_imagenes`.

6.3. Consumo en frontend

- El listado (`VehicleList` / `VehicleCard`) pide al backend los vehículos con su **imagen de perfil** ya resuelta en el SELECT.

- El detalle (`VehicleDetailDialog`) solicita también la lista completa de imágenes para ese vehículo y arma una galería.

Ventajas de usar ImageKit:

- Optimización automática de imágenes (tamaño, calidad).

- URLs con transformación (si se quieren aplicar recortes o tamaños según dispositivo).

- Menos carga en el servidor Node y en la base de datos.

- Menos problemas de permisos y almacenamiento local.

---

7. Ventas, formas de pago y cuotas

Aunque la consigna solo pedía “al menos dos tipos de forma de pago”, en el proyecto se modeló un sistema más completo:

- **Formas de pago**:
    - Efectivo.
    - Transferencia.
    - Posibilidad de otros medios (ej. tarjeta, financiaciones).
- **Cuotas**:

    - `SaleCuotasDialog` en frontend para que el usuario (vendedor) pueda definir una cantidad de cuotas, montos, etc.
    - El `SaleCreateDialog` recibe el `id_vehiculo` y el cliente, y a partir de ahí:
        - Registra la venta en la tabla `ventas`.
        - Guarda la información de pagos/cuotas (según el diseño que se haya implementado).
- Se contempló una bandera de **"vehículo entregado"** o estado equivalente, para distinguir entre reservado, vendido, entregado, etc.

---

8. Autenticación, roles y permisos

El sistema incorpora una capa completa de autenticación y autorización basada en **JSON Web Tokens (JWT)**.
El backend genera un token al iniciar sesión y el frontend lo almacena para acceder a las áreas protegidas.

8.1 Autenticación

- **Login de usuario** mediante email/contraseña.
- El backend genera un **JWT** firmado que contiene `id_usuario`, `nombre` y `id_rol`.

8.2 Áreas públicas

- Cualquier usuario (con o sin login) puede:
    - Ver el **listado de vehículos**
    - Ver el **detalle de un vehículo**
    - Visualizar imágenes y características

Esto permite navegar el catálogo sin necesidad de cuenta.

8.3 Áreas protegidas

Requieren usuario autenticado:

- **Comprar un vehículo** (botón Comprar → abre SaleCreateDialog)
- **Dashboard de ventas** (Admin y Vendedor)
- **Gestión de vehículos** (crear/editar/baja lógica)
- **Mis compras / Mis movimientos** (cualquier usuario logueado)

8.4 Roles

El sistema utiliza tres roles principales:

Admin (id_rol = 1) | Vendedor (id_rol = 2)

- Puede crear/editar/borrar vehículos
- Administrar ventas
- Ver dashboard completo
- Gestionar pagos y cuotas
- Ver movimientos generales del sistema

Cliente / Usuario (id_rol = 3)

- Puede:
  - **Comprar vehículos**
  - Acceder a **“Mis compras”** (ventas propias)
  - Acceder a **“Mis movimientos”** (pagos y cuotas propias)
- No puede acceder al dashboard global ni gestionar vehículos

8.5 Rutas privadas (frontend)

El frontend protege las rutas mediante `PrivateRoute`, que valida:

1. Que el usuario esté logueado
2. Que el rol tenga permisos según la vista
    -

---

9. Infraestructura con Docker

Decisión importante: dejar el proyecto listo para levantarse con un solo comando:

- Archivo `docker-compose.yml` con servicios:
    - db (MySQL) con:
        - Volumen para persistencia de datos.
        - Carpeta `init` para scripts SQL iniciales.
    - `phpmyadmin` para facilitar inspección de datos.
    - `backend`:
        - Construido desde Dockerfile propio.
        - Dependiendo de la base de datos.
    - `frontend`:
        - Construido desde Dockerfile propio.
- Beneficios:

Cualquiera puede clonar el repo y levantar todo el entorno con:

docker compose up --build

- Se asegura que la versión de Node y MySQL que se usan sean controladas.
- Simplifica la corrección en la prueba técnica, porque el evaluador no tiene que instalar nada aparte.

---

10. Buenas prácticas adicionales

- **Variables de entorno**:
  - Credenciales de DB, puertos y otros parámetros se leen desde variables de entorno y no se hardcodean en el código.
- **Manejo de errores**:
  - Respuestas JSON claras en caso de error (`message`, `details` cuando corresponda).
  - Logs en backend para poder depurar problemas.
- **Validaciones**:
  - En backend, validaciones de tipos y presencia de campos clave antes de llamar a los stored procedures.
  - En frontend, validaciones básicas de formularios (campos requeridos, formatos numéricos para precio, km, etc.).

---

11. Posibles mejoras futuras

- Ajustar aún más el uso de **transformaciones de ImageKit** (thumbnails, recortes específicos para mobile/desktop).
- Agregar versiones de baja resolución para `lazy loading` o placeholders.
- Soporte de drag-and-drop más avanzado para reordenar imágenes, eliminar, editar en el panel de administración.

- Reportes más avanzados de ventas por período, por vendedor, por forma de pago.
- Panel de administración más completo para gestionar catálogos (marcas, colores, tipos, etc.).