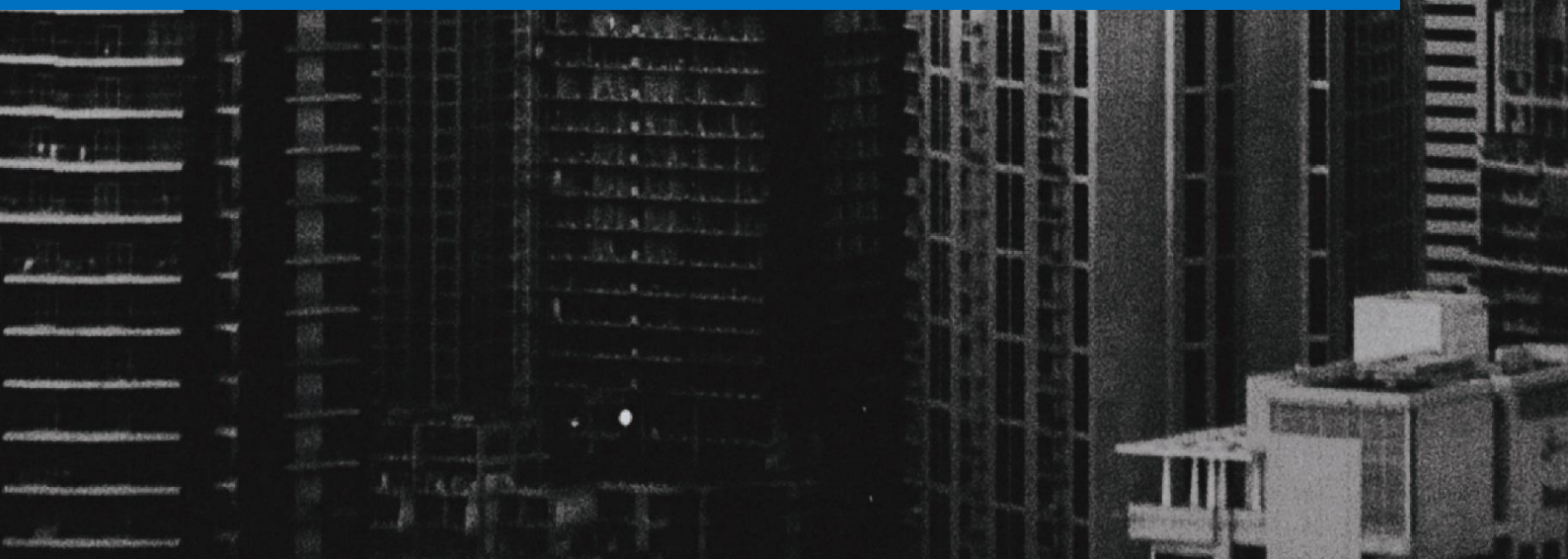
PROYECTO 4

CC3088 - Bases de Datos 1

Integrantes

Dulce Rebeca Ambrosio Jiménez	231143
Daniel Alejandro Chet Delgado	231177
Javier Alexander Linares Chang	231135
Gadiel Amir Ocaña Veliz	231270
Cristian Sebastián Túnchez Castellanos	231359



Sistema de alquiler de vehículos

Objetivo del Proyecto

Este proyecto tiene como propósito el diseño e implementación de un sistema completo de alquiler de vehículos, integrando modelado de datos, validaciones, vistas SQL, reportería y el uso de un ORM sin escribir SQL crudo. Se busca reflejar un sistema real que administre el flujo de clientes, vehículos, reservas, contratos, pagos y mantenimientos.

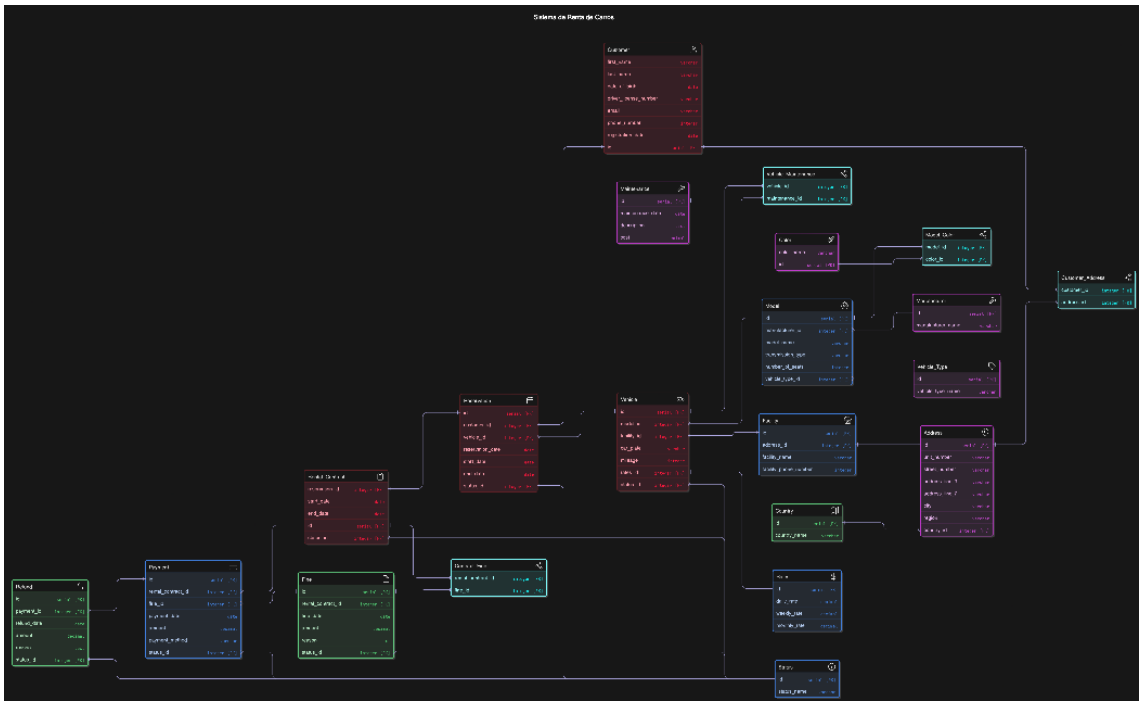
Descripción General del Proyecto

Este proyecto representa la implementación final de un sistema completo de alquiler de vehículos, abordando todos los aspectos esenciales desde la gestión de clientes y flota, hasta reportes avanzados y control de pagos. Fue desarrollado con la siguiente estructura: backend en FastAPI con SQLAlchemy, frontend en React, y base de datos PostgreSQL esto dockerizado.

El desarrollo se basó en ORM, sin uso de SQL crudo, cumpliendo los requerimientos del proyecto y simulando condiciones reales de un sistema de gestión empresarial.

Diseño de la Base de Datos

El diagrama ER está compuesto por más de 20 tablas relacionadas de forma normalizada hasta Tercera Forma Normal (3FN). Se implementaron relaciones 1:N y N:M, y se representaron atributos derivados o multivaluados a través de relaciones auxiliares. La base del modelo gira en torno a las entidades Customer, Vehicle, Reservation y Rental_Contract, que representan el flujo natural del negocio.



[Vista detallada del Diagrama ER clic aquí]

Entidades Principales

- **Customer:** Guarda los datos del cliente, incluyendo información personal, licencia, contacto y fecha de registro.
- **Vehicle:** Cada vehículo se relaciona con un modelo específico y una sede (Facility). También se incluye la placa, kilometraje y estado.
- **Reservation:** Representa una solicitud de renta por parte de un cliente para un vehículo, con fechas de inicio y fin.
- **Rental_Contract:** Formaliza la transacción después de la reserva, añadiendo estado, fechas efectivas y asociándose a multas y pagos.

Relaciones Relevantes

- **1:N** entre Customer y Reservation, Reservation y Rental_Contract, Rental_Contract y Payment, Rental_Contract y Fine.

- **N:M entre:**
    - Vehicle y Maintenance (a través de Vehicle_Maintenance).
    - Rental_Contract y Fine (a través de Contract_Fine).
    - Model y Color (por medio de Model_Color), para permitir modelos multicolor.
- **Direccionalidad clara:** todas las claves foráneas apuntan desde las entidades operativas hacia sus definiciones base, lo cual simplifica las operaciones de inserción y eliminación.

Atributos Derivados o Multivaluados

- El estado del contrato y de la reserva se gestiona mediante una tabla separada Status, centralizando el control de estados válidos.
- La relación Model_Color permite definir que un mismo modelo puede fabricarse en varios colores, evitando duplicidad de datos.

Normalización

Se aplicó un proceso riguroso de normalización:

- Eliminamos repeticiones de datos (por ejemplo, dirección del cliente dividida entre Address y Customer_Address).
- Separamos atributos compuestos (como Manufacturer, Vehicle_Type, etc).
- Modelamos datos dependientes funcionales como tarifas en Rates, como entidades independientes.

Tipos de Datos personalizados

- **Payment**
    - Method: enum(cash, card, transfer)
- **Model**
    - Transmission_type: enum(manual, automatic, continuamente variable)
    - Capacity: enum(2, 5, 7, 8)

Informe Técnico de la API – Sistema de Renta de Vehículos

1. Introducción

Como parte de la documentación de este proyecto, queremos describir la arquitectura, funcionalidad y alcance técnico de la API desarrollada para el sistema de gestión de renta de vehículos. La solución fue construida utilizando el framework FastAPI junto a SQLAlchemy como ORM, integrando principios de modularidad, validación robusta y una clara separación entre lógica de negocio, persistencia y presentación.

2. Especificaciones Generales

- **Framework**: FastAPI
- **Versión de la API**: 0.1.0
- **Especificación OpenAPI**: 3.1.0
- **Base de datos**: PostgreSQL
- **ORM**: SQLAlchemy
- **Frontend asociado**: React + Vite + Material UI (para reportes)

La API fue diseñada como un backend RESTful con capacidad de interacción directa a través de HTTP, y responde en formato JSON.

3. Funcionalidad CRUD

Se implementaron tres módulos principales con operaciones completas de creación, lectura, actualización y eliminación (CRUD), cumpliendo estándares REST y validaciones tanto a nivel de modelo como de aplicación.

3.1 Módulo de Clientes

Método	Ruta	Descripción
GET	/customers/	Obtener lista paginada de todos los clientes
GET	/customers/{customer_id}	Obtener datos de un cliente específico por ID
POST	/customers/	Registrar un nuevo cliente
PUT	/customers/{customer_id}	Actualizar información de un cliente existente
DELETE	/customers/{customer_id}	Eliminar un cliente del sistema

Este módulo permite la administración completa del registro de clientes, incluyendo validaciones de correo electrónico, teléfono y número de licencia.

3.2 Módulo de Vehículos

Método	Ruta	Descripción
GET	/vehicles/	Listar todos los vehículos registrados
GET	/vehicles/{vehicle_id}	Consultar información de un vehículo específico
POST	/vehicles/	Registrar un nuevo vehículo
PUT	/vehicles/{vehicle_id}	Modificar datos de un vehículo
DELETE	/vehicles/{vehicle_id}	Eliminar un vehículo del sistema

Este módulo gestiona la flota disponible, controlando atributos como placa, kilometraje, tarifas asociadas y estado operativo.

3.3 Módulo de Reservas

Método	Ruta	Descripción
GET	/reservations/	Obtener todas las reservas registradas
GET	/reservations/{reservation_id}	Consultar una reserva específica por ID
POST	/reservations/	Crear una nueva reserva
PUT	/reservations/{reservation_id}	Modificar fechas o estado de una reserva existente
DELETE	/reservations/{reservation_id}	Eliminar una reserva del sistema

Este módulo enlaza clientes con vehículos disponibles para un período determinado, incluyendo lógica de validación de fechas y actualizaciones automáticas mediante **triggers**.

4. Endpoints de Reportes Analíticos

Los reportes analíticos son consumidos desde vistas SQL complejas, diseñadas para facilitar análisis operativos, financieros y de mantenimiento.

4.1 Resumen de Reservas

- **Endpoint**: /analytics/reservations
- **Vista SQL**: view_reservation_summary
- **Filtros**: from_date, to_date, status
- **Contenido**: ID de reserva, cliente, vehículo y estado textual.

4.2 Historial de Mantenimientos

- **Endpoint**: /analytics/maintenance
- **Vista SQL**: view_vehicle_maintenance_history
- **Filtros**: car_plate, min_cost, max_cost, from_date, to_date
- **Contenido**: Fecha, descripción y costo por vehículo.

4.3 Ingresos por Contrato

- **Endpoint**: /analytics/contracts/income
- **Vista SQL**: view_total_income_per_contract
- **Filtros**: contract_id, min_total_income, max_total_income, min_fines, max_fines
- **Contenido**: Pagos, multas e ingreso total por contrato.

5. Funciones SQL y Utilidades

Se integraron funciones SQL definidas por el usuario que encapsulan lógica de negocio crítica para la plataforma:

5.1 Cálculo de Días de Reserva

- **Función**: calculate_rental_days(start_date, end_date)
- **Endpoint**: /utils/rental-days
- **Propósito**: Retorna el número de días de una reserva de forma precisa.

5.2 Estimación de Costo de Reserva

- **Función**: calculate_reservation_cost(vehicle_id, start_date, end_date)
- **Endpoint**: /utils/estimate-cost
- **Propósito**: Devuelve el costo estimado de una reserva basado en tarifas diarias, semanales y mensuales.

6. Auditoría y Seguridad de Datos

Para garantizar trazabilidad y reforzar la integridad del sistema, se implementaron los siguientes triggers:

- **Trigger 1**: Validación automática de fechas al crear o modificar reservas.
- **Trigger 2**: Cambio automático del estado de un vehículo al confirmarse una reserva.
- **Trigger 3**: Registro de auditoría en la tabla payment_log cada vez que se inserta un pago.

La API expone el endpoint **/logs/payments** para consultar dichos registros de auditoría de forma controlada.

7. Validaciones

La validación de los datos se maneja tanto en el nivel de base de datos como en el backend:

- **SQL**: Uso de NOT NULL, UNIQUE, CHECK, DEFAULT, y funciones/triggers.
- **FastAPI + Pydantic**: Validación estructural (tipo, longitud), validación cruzada y control de errores con HTTPException.

8. Reportes y Exportaciones

El frontend desarrollado para esta API permite:

- Visualización clara de datos en tablas organizadas.
- Aplicación de filtros significativos por cada vista analítica.
- Exportación directa de resultados a formato .csv mediante botones integrados.
- Navegación estructurada entre reportes mediante pestañas.

9. Datos de Prueba

Se cargaron más de 1000 registros entre todas las tablas del sistema, con datos coherentes y variados. Estos se cargan automáticamente con el script data.sql incluido en el proyecto.

10. Anexos

Repositorio en GitHub (Controlador de Versiones)

[Clic en la imagen o aquí]

Reflexión

1. **¿Cuál fue el aporte técnico de cada miembro del equipo?**

Este proyecto fue la continuación del Proyecto 3, en el cual ya habíamos desarrollado el modelo de base de los datos, restricciones, triggers iniciales y carga de datos. Para el Proyecto 4 nos enfocamos en seguir con este trabajo, cumpliendo los nuevos requerimientos como el uso obligatorio de un ORM, definición de tipos personalizados, implementación de vistas SQL y funciones, así como la generación de una base de datos más robusta con más de mil registros de prueba.

Javier y Cristian se encargaron de estructurar toda la parte del backend con SQLAlchemy, trasladando el modelo anterior a código ORM y generando los scripts automatizados. Dulce y Gadiel se enfocaron en mejorar las restricciones y mantener la consistencia de los datos mientras se expandía la base (poblar la base de datos). Dulce se encargó de actualizar y justificar el diagrama ER de acuerdo con los nuevos requerimientos. Daniel trabajó en los triggers que hacían falta y en las nuevas funciones SQL que se pedían. Y finalmente todo el grupo realizó la documentación del proyecto por lo que se puede decir que cada miembro contribuyó sobre la base ya existente y nos enfocamos en cumplir lo que el Proyecto 4 solicitaba.

2. **¿Qué decisiones estructurales se tomaron en el modelo de datos y por qué?**

Decidimos modelar el sistema de forma modular para evitar duplicidades y facilitar su mantenimiento. Separar tablas como Address, Rates, Status, Vehicle_Type o Manufacturer, lo que nos permitió tener un sistema más limpio, reutilizable y escalable. También optamos por modelar relaciones N:M mediante tablas intermedias como Vehicle_Maintenance, Contract_Fine ya que esto nos da más flexibilidad si en el futuro se agregan atributos extra en esas relaciones.

3. **¿Qué criterios siguieron para aplicar la normalización?**

Aplicamos la normalización hasta 3FN y BCNF con base en tres criterios principales: evitar datos redundantes, facilitar actualizaciones y asegurar integridad. Por ejemplo, separamos la dirección del cliente a una entidad aparte para no repetir campos, y definimos entidades únicas para Status y Payment_Method para centralizar esos valores. También nos guiamos por dependencias funcionales claras para así no tener campos calculados dentro de una misma tabla.

4. **¿Cómo estructuraron los tipos personalizados y para qué los usaron?**

Utilizamos ENUM directamente en PostgreSQL para controlar valores como tipo de transmisión, estado del contrato/reserva y método de pago. Esto nos permitió validar desde la base y evitar registros con valores inconsistentes. Ya que usar ENUMs hace que el sistema sea más fácil de mantener y mejorar la legibilidad del modelo de datos tanto para nosotros como para otros desarrolladores.

5. **¿Qué beneficios encontraron al usar vistas para el índice?**

Las vistas nos ayudaron a simplificar la lógica del backend. Por ejemplo, en lugar de hacer múltiples JOINs en cada endpoint, usamos vistas para mostrar directamente la información necesaria del cliente, vehículo, reserva o contrato. Tomando en cuenta que mejora la eficiencia en consultas, reduce errores en la lógica y nos permite exponer solo lo necesario a nivel de aplicación, también facilitó bastante las pruebas en Postman.

6. **¿Cómo se aseguraron de evitar duplicidad de datos?**

Nos enfocamos en tres cosas: normalización, restricciones únicas (UNIQUE), y validaciones en triggers y funciones. Por ejemplo, en la tabla Customer el campo driving_license_number tiene una restricción única, igual que la placa en Vehicle. También se usaron CHECKS para limitar los valores de estados, y funciones para asegurar consistencia en cálculos o rangos de fechas.

7. **¿Qué reglas de negocio implementaron como restricciones y por qué?**

Implementamos diferentas reglas como: una reserva no puede tener fecha de fin anterior a la de inicio, que el monto de una multa no sea negativo y que los pagos estén asociados a contratos válidos. Estas reglas se colocaron tanto como restricciones SQL (CHECK, NOT NULL) como en triggers o validaciones en el backend, para evitar y garantizar que no se rompa la lógica del negocio sin importar desde dónde se interactúe con el sistema.

8. **¿Qué trigger resultó más útil en el sistema? Justifica.**

El trigger más útil fue el que calcula automáticamente el total a pagar al crear un contrato, usando las fechas y las tarifas ya que evita tener que pasar ese dato desde la aplicación, lo que reduce errores humanos, además permite tener reportes más confiables, ya que el cálculo es uniforme y se ejecuta directamente desde la base de datos en el momento correcto.

9. **¿Cuáles fueron las validaciones más complejas y cómo las resolvieron?**

La validación más complicada fue evitar que un mismo vehículo tenga reservas solapadas ya que se necesita comparar fechas de inicio y fin contra reservas existentes, y eso lo resolvimos con lógica en el backend combinada con una función SQL de apoyo. También fue un reto manejar la relación entre multas, pagos y reembolsos, ya que depende del estado de cada contrato. Para lo cual usamos funciones y triggers para asegurar la integridad en estos casos.

10. **¿Qué compromisos hicieron entre diseño ideal y rendimiento?**

Optamos por enfocarnos solo en los requerimientos obligatorios, priorizando un diseño claro y normalizado. No implementamos reportes avanzados para no complicar el sistema y preferimos mantener la lógica crítica en funciones SQL, aunque eso implicara un pequeño costo en rendimiento, a cambio de asegurar consistencia y facilitar el mantenimiento.