**Fleet Nexa: On-Demand Logistics Platform for Goods Transportation**

**INTRODUCTION**

Fleet Nexa is a modern, on-demand logistics platform designed to streamline the transportation of goods for both users and drivers. Built with microservice architecture, Fleet Nexa enables real-time booking, live tracking, dynamic pricing, and seamless fleet management. By utilizing cutting-edge technologies such as WebSocket, Redis, Kafka, and PostgreSQL, the platform provides fast, reliable, and scalable services to handle high-volume traffic while ensuring real-time data flow.

With its intuitive interface and intelligent backend, Fleet Nexa is tailored to efficiently connect users to a vast fleet of drivers, ensuring timely deliveries and transparent communication at every step of the process. From booking a vehicle to real-time tracking and feedback, every aspect is designed to enhance the user experience while keeping operations smooth for drivers and administrators alike.

**SYSTEM DESIGN**

* **Microservice Architecture**: Chose a modular, service-based design to manage complexity and scale individual services independently. This allows each service, such as booking, real-time tracking, and admin, to operate autonomously, promoting flexibility and fault isolation.
* **Real-Time Location and Web Sockets**: For live tracking, we integrated WebSocket to ensure continuous, low-latency communication between users and drivers. This approach keeps users engaged and up to date on deliveries in real-time.
* **Redis for Live Location Storage**: Redis was selected for its low-latency and high-speed read-write capabilities. Storing real-time GPS data in Redis ensures quick retrieval, crucial for real-time updates.
* **Database Sharding**: Implemented sharding based on the user's email to distribute the data across multiple databases, reducing bottlenecks, and ensuring faster data access for a high volume of users.
* **Pricing Service with Dynamic Calculation**: We developed a flexible pricing service that dynamically adjusts based on factors such as distance, vehicle type, and time slot, ensuring scalability as the system grows and global traffic increases.
* **Scheduled Bookings**: Added support for future bookings to accommodate user preferences. This introduces additional considerations for efficient scheduling and resource allocation.

A black and white image of several hexagons

Description automatically generated

**TECH STACK**

* Next.js: Ensures fast, server-side rendered pages for a smooth user experience.
* TypeScript: Enhances code quality and scalability with type safety.
* Node.js & Express.js: Efficiently handles multiple concurrent requests with a flexible API structure.
* PostgreSQL: Provides reliable, persistent storage for users, bookings, drivers, and vehicles.
* Redis: Ensures fast reads and writes for real-time tracking with live location data.
* Kafka: Manages asynchronous communication between services for seamless event-driven processing.
* WebSockets: Facilitates real-time updates between users and drivers for location tracking and job status changes.
* Supabase: Simplifies database operations and authentication with seamless integration into PostgreSQL.

**FEATURES IMPLEMENTED**

1. **User Booking System**

* Booking Process: Users can easily book deliveries through a user-friendly interface.
* Driver Acceptance: Upon receiving a booking request, the delivery driver can view and accept the request.

2. **Location Management**

* Pickup and Drop-off Locations: Drivers initially see only the pickup location. After accepting the request, they gain access to the drop-off location.
* Real-Time Location Tracking: Both users and drivers can track each other's real-time locations, ensuring transparency and safety during the delivery process.
* Shortest Route Guidance: Drivers receive the shortest route to reach the pickup and drop-off locations efficiently.

3. **Booking History**

* View All Bookings: Users and drivers can access their booking history, providing a comprehensive overview of past deliveries and interactions.
* Driver Ratings: After successful deliveries, users can rate the driver based on their experience, contributing to quality assurance.

4. **Pricing Model**

* Dynamic Cost Estimation: The system calculates estimated delivery costs based on various factors:
  + Distance between pickup and drop-off locations
  + Type of vehicle selected
  + Vehicle capacity
  + Time of delivery

5. **Admin Management**

* Fleet Management: Admins can manage the entire fleet, including adding and removing vehicles and drivers.
* Booking Oversight: Admins have access to view all drivers' bookings and ratings, ensuring operational oversight.
* Availability Management: Admins can mark vehicles and drivers unavailable, managing resources effectively during peak times or maintenance.

6. **Real-Time Tracking Service**

* Live Location Storage: Redis is utilized for storing live location data, allowing for rapid updates and retrieval of current location information. This is particularly beneficial for frequently changing data.

7. **Database Structure**

* The system maintains five core tables:
  + Admins: For administrative user management.
  + Bookings: To track all delivery requests and statuses.
  + Drivers: Information and management of all drivers in the fleet.
  + Users: To manage end-user information and profiles.
  + Vehicles: Details and specifications of all vehicles in the fleet.
  + Payments: Details and specifications of all the payments.

8. **User Interface**

* Polished and Elegant UI: The user interface is designed to be intuitive and visually appealing, enhancing the overall user experience.

9. **Microservice Architecture**

* The system is built with microservice architecture.
* An API Gateway is implemented to route requests efficiently.
* Kafka is used as a messaging queue to facilitate seamless communication between services.
* Health Check Service is made to check in-active services.

**SCALING FLEET NEXA**

1. **Horizontal scaling and microservices:**

* Scale services independently by running multiple instances of each microservice (e.g., booking, real-time tracking) to handle increased traffic.

2. **Database sharding and master-slave architecture:**

* Distribute the database load through sharding (based on email ID) and use a master-slave setup for high availability and read-heavy operations.

3. **Caching with Redis:**

* Use Redis to cache frequently accessed data, such as live locations, to reduce database load and speed up read/write operations for real-time data.

4. **API gateway with load balancing:**

* The API gateway balances incoming requests across service instances, ensuring no single instance is overwhelmed and managing traffic efficiently.

5. **Message queuing with Kafka:**

* Kafka helps with asynchronous communication between services, allowing the system to handle traffic spikes without overloading any single service.

6. **Auto-scaling and service replication:**

* Implement auto-scaling to adjust the number of running instances based on traffic, ensuring scalability and resource efficiency, while replicating services for redundancy.

7. **Connection pooling and rate limiting:**

* Manage database connections more efficiently through connection pooling and use rate limiting to protect APIs from being overwhelmed by high traffic.

8. **Monitoring, alerts, and backup:**

* Monitor system health with tools like Prometheus, set up alerts for traffic surges or resource usage issues, and maintain backup systems to handle failures or disaster recovery.

**ER DIAGRAM**

A black background with white text

Description automatically generated