System Design Day 11: Ride-Sharing system

🚗 Ride-Sharing System Design Summary

✅ Core Functional Requirements

- Rider selects a pickup location via map.
- Sees ETA and price.
- Can pay for the ride.
- Gets matched to a driver.
- Receives real-time driver updates.

🧱 High-Level Architecture Components

1. **Clients**: Riders and drivers (mobile/web).
2. **APIs**: Rider & Driver APIs behind Load Balancers; support horizontal scaling.
3. **Databases**:
    - **User DB**: Info about riders & drivers.
    - **Trips DB**: Trip status, route, fare, etc.
    - Use **sharding** on `user_id` and `trip_id` for scalability.
    - Use **global indexes** (by driver_id/rider_id) for fast queries.
4. **Event Bus (Kafka)**:
    - Enables decoupling and async communication.
    - Used for real-time data updates (e.g., driver location, ride requests).
5. **Map Service (Mapbox/Google Maps)**:
    - Serves map tiles, geocoding, directions.
6. **Payment Integration (e.g., Stripe)**:

- Stripe processes payment and invokes a webhook.
- Webhook pushes event to Kafka for ride updates.

7. **Pricing Service**:
   - Implements **dynamic pricing** (surge based on demand).
   - Uses **Spark Structured Streaming** on Kafka events → Redis cache.
8. **Matching Service**:
   - Matches riders with nearby drivers.
   - Uses **Uber's H3 hexagonal geospatial index** for location clustering.

🔄 Real-Time Data Flow

- Use **WebSockets** (vs. polling) for location updates.
- Driver location pushed to rider in real time.
- Kafka event bus distributes updates across services.

💡 Key Design Considerations

- **Scalability**:
  - Load-balanced APIs.
  - Horizontally sharded DB.
- **Fault Tolerance**:
  - Event-driven system allows for async retries.
- **Performance**:
  - Redis cache for pricing and demand.
  - H3 for fast location-based lookups.
- **Extensibility**:
  - Plug-and-play with external services like Stripe and Mapbox.

📊 Example: Ride Lifecycle

1. Rider opens app → sees map via Mapbox.
2. App queries ETA via Ride Service.

3. Price fetched from Pricing API (uses Redis + Spark).
4. Rider books and pays via Stripe → webhook fires → event added to Kafka.
5. Matching Service finds nearby driver using H3 cells.
6. Driver accepts → both parties notified → real-time location sent over WebSocket.

🔍 Optimization Topics (Not Fully Covered)

- Redundancy and failure recovery.
- ETA prediction algorithms.
- Geo-shard placement strategies.
- Data analytics and BI pipelines.

This system showcases **event-driven microservices**, **real-time communication**, **distributed data processing**, and **3rd-party service integration** — making it scalable, modular, and robust for production.s