Food Delivery App

Step 1: Understand the Problem and Define Design Scope

Q&A Process to Establish Scope:

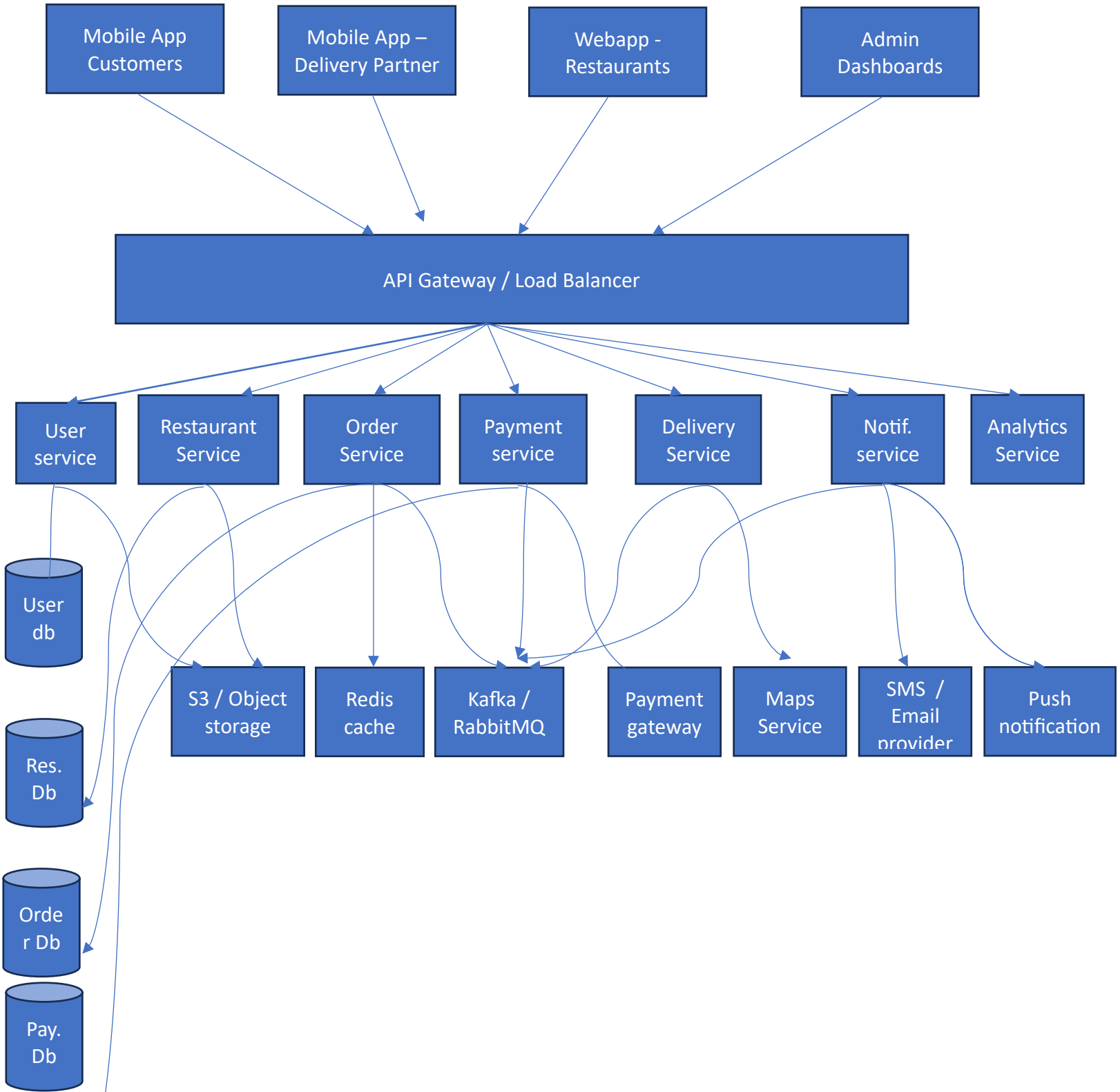
1. **Platform Requirement**:
    - **Q**: Is the service for mobile, web, or both?
    - **A**: Both mobile and web interfaces.
2. **User Scale**:
    - **Q**: What is the app scale? Is it a startup or a large-scale system?
    - **A**: Designed to handle millions of daily active users.
3. **Features Required**:
    - **Q**: What features are needed in the system?
    - **A**: Restaurant search and ordering, real-time delivery tracking, online/offline status, notifications, multi-device access, and payment processing.
4. **Data Storage Requirements**:
    - **Q**: How long do we store order history?
    - **A**: Indefinitely, for users to access past orders, loyalty programs, etc.
5. **Security & Compliance**:
    - **Q**: Do we need secure handling for sensitive data?
    - **A**: Yes, especially for payment and user data.

Functional Requirements:

1. User and restaurant profiles and authentication.
2. Browse and search restaurants, menus, and place orders.
3. Real-time order tracking.
4. Push notifications for updates.
5. Payment processing, including refunds.
6. High availability and fault tolerance.

Step 2: High-Level Design

The design is based on a microservices architecture, with services communicating via HTTP and WebSockets where necessary (e.g., for real-time delivery tracking).



Client Layer

- **Customer Mobile/Web App**: Enables browsing, ordering, and delivery tracking.
- **Delivery Partner App**: Provides order acceptance, navigation, and live updates.
- **Restaurant Dashboard**: Allows restaurants to manage orders, update availability, and view analytics.
- **Admin Dashboard**: Used for system monitoring, escalation handling, and user/restaurant management.

API Gateway

- **Load Balancing and Routing**: Routes incoming requests to appropriate services.
- **Security and Rate Limiting**: Protects services from excessive traffic.
- **API Versioning**: Enables backward compatibility for different app versions.
- **Transformation**: Converts data formats for consistent communication across services.

Service Layer Design

Each service is stateless, enabling independent scalability and deployment:

1. **Auth Service**: Manages authentication and authorization (OAuth, JWT).
2. **User Service**: Manages user profiles, preferences, and authentication.
3. **Restaurant Service**: Handles restaurant data, including menus, hours, and availability.
4. **Order Service**: Tracks order status, updates, and lifecycle.
5. **Payment Service**: Manages payment processing and refunds through external gateways.
6. **Delivery Service**: Assigns delivery partners and optimizes routes based on geolocation.
7. **Notification Service**: Sends order updates, status changes, and promotions through push notifications.
8. **Analytics Service**: Analyzes usage patterns, peak times, and delivery performance for business insights.

Message Queue (Kafka/RabbitMQ)

- **Asynchronous Messaging**: Decouples communication between services for events like order status changes, notifications, and delivery updates.
- **Event Handling**: Manages delivery status changes, analytics updates, and in-app notifications for a responsive user experience.

Data Layer Design

The data layer is designed with scalability, partitioning, and data consistency in mind:

- **Databases**:
  - **User Database**: Stores customer profiles and preferences.
  - **Order Database**: Holds order details, statuses, and timestamps.
  - **Restaurant Database**: Stores menus, operating hours, and restaurant metadata.
  - **Payment Database**: Manages transactions, refunds, and billing details.
- **Sharding**: Horizontal partitioning of data based on user location or restaurant ID for scalability.
- **Read Replicas**: Maintains read replicas for read-heavy databases to optimize performance.

Caching Layer (Redis)

- **Cached Data**:
  - **Restaurant Menus**: Frequently accessed by users.
  - **User Profiles**: Reduces load on the database.
  - **Active Orders**: Ensures fast updates for delivery tracking.
- **TTL Mechanisms**: Ensures data freshness for frequently changing information, like order statuses.

Step 3: Detailed Component Design

Communication Channels

- **WebSocket**: Real-time, bidirectional channel for critical updates, like live delivery tracking.
- **HTTP**: Used for non-real-time services like login, profile updates, and payment processing.

Stateful and Stateless Services

- **Stateless Services**: Scalable and independently deployable, including User, Restaurant, and Order services.
- **Stateful Service (Chat Server)**: Used for continuous communication with delivery partners, leveraging WebSockets to maintain stateful connections for live tracking.

Step 4: Scalability and Reliability

Horizontal Scaling:

- **Microservices**: Each service scales independently.
- **Database Sharding**: Scales data horizontally.
- **Auto-scaling**: Services like delivery and notification dynamically scale based on demand.

High Availability:

- **Fault Tolerance**: Distributed services, load balancing, and backups.
- **Failover**: Automatic failover for high-priority services to maintain uptime.

Data Consistency and Storage

- **KV Store (e.g., Cassandra)**: Stores chat and delivery history. Optimized for fast access with low latency, especially for real-time delivery tracking.
- **Object Storage**: Holds images, invoices, and other media for lightweight access.
- **Data Consistency**: Ensures eventual consistency for non-critical data and implements the SAGA pattern for distributed transactions between services like Order and Payment.

Data Models

- **Order Table**:
  - order_id (PK), user_id, restaurant_id, status, total_price, created_at
- **Delivery Table**:
  - delivery_id (PK), order_id, delivery_partner_id, status, assigned_at, completed_at
- **User Table**:
  - user_id (PK), name, email, phone, address, created_at

Step 5: High-Level Flow Deep Dive

Service Discovery

- **Load Balancing**: Recommends servers based on proximity and server load. For example, a service discovery mechanism (e.g., Zookeeper) directs clients to the optimal service endpoint.

Message Flow

- **Order Lifecycle**: Customer places an order → Restaurant confirms → Delivery partner assigned → Order tracked to completion.

- **Status Updates**: Use WebSockets to send updates to customers and restaurant staff in real time.

Delivery Sync Across Devices

- Each device maintains a consistent state, using WebSocket connections to receive updates. This is achieved by storing the latest message_id to ensure message sync across mobile and desktop.

Step 6: Real-Time Presence and Notifications

1. **Online Presence**:
    - Updates status when users log in, displayed as online/offline in the app.
    - Uses publish-subscribe for presence updates, so users can see which delivery partners or restaurants are available.
2. **Notifications**:
    - **Push Notifications**: For order updates, new offers, and promotions.
    - **SMS/Email**: For critical updates (e.g., order confirmation, delivery complete).
3. **Heartbeat Mechanism**: Keeps delivery partners connected and updates their presence. If no heartbeat is received after a set time, the partner is marked offline.

Step 7: Wrap-Up & Future Considerations

1. **Media Support**: Enable sharing of photos (e.g., delivery proof images).
2. **End-to-End Encryption**: Ensure secure communication for sensitive data.
3. **Client-Side Caching**: Reduces server load and improves data retrieval speed.
4. **Error Handling**:
    - Retry mechanisms for message delivery and notifications.
    - Automatic failover to reconnect to the chat server if a server goes down.