happy (2 months ago)

My Solution for Design Food Delivery Service | Score: 89

Solution

89%

Drawing

87%

108 0 0 0

n. / Non-Fun. Requirements

ınctional Requirements

- **User Registration and Authentication:** Users can sign up and log in via email, phone, or social media with secure authentication.
- **Restaurant Listings:** Show restaurants based on location with details like menu, ratings, and reviews.
- **Order Placement:** Users can add items to a cart and customize orders.
- **Payment Processing:** Integrate multiple secure payment gateways.
- **Order Tracking:** Real-time order status updates and notifications.
- **Customer Support:** In-app chat or call support for queries and feedback management.
- **Review and Rating System:** Users can rate and review restaurants and deliveries.

ɔn-Functional Requirements

- **Scalability:** Handle many concurrent users and orders with a scalable architecture.
- **Performance:** Fast response times and efficient data processing.
- **Reliability:** High availability with failover mechanisms.
- **Security:** Encrypt user data and implement robust authentication.
- **Flexibility:** Modular design for easy feature additions.
- **Accessibility:** Comply with accessibility standards for users with disabilities.

ıffic Estimation and Data Calculation

·affic Estimation and Data Calculation

ssumptions

- **User Base:** 1 million active users
- **Daily Active Users (DAU):** 100,000 users (10% of user base)

- **Peak Traffic:** 20,000 users (20% of DAU)
- **Average Orders per User per Day:** 1 order
- **Average Items per Order:** 3 items

rite Flow

- **Order Placements:** 100,000 orders/day
- **Reviews and Ratings:** 50,000 reviews/day

ead Flow

- **Menu Browsing:** 500,000 requests/day
- **Order Tracking:** 300,000 requests/day

ata Storage

- **User Data:** 1 GB (1 million users x 1 KB)
- **Order Data:** 182.5 GB/year (100,000 orders/day x 5 KB x 365 days)
- **Review Data:** 9.125 GB/year (50,000 reviews/day x 0.5 KB x 365 days)

etwork Bandwidth

- **Peak Hour API Requests:** 400,000 requests (20,000 users x 20 API calls)
- **Data Transfer During Peak:** 400 MB (400,000 requests x 1 KB)

ummary

- **Total Daily API Requests:** 2 million requests
- **Total Daily Data Transfer:** 2 GB (2 million requests x 1 KB)
- **Annual Data Storage Requirement:** ~200 GB/year

I Design

ser Registration and Authentication

- **Endpoint**: `/api/v1/users/register`
  - **Method**: POST
  - **Request**: `{ "email": "string", "password": "string", "phone": "string" }`
  - **Response**: `{ "userId": "string", "message": "User registered successfully" }`
- **Endpoint**: `/api/v1/users/login`
  - **Method**: POST
  - **Request**: `{ "email": "string", "password": "string" }`
  - **Response**: `{ "token": "string", "message": "Login successful" }`

estaurant Listings

- **Endpoint**: `/api/v1/restaurants`
  - **Method**: GET
  - **Request**: `?location=string&cuisine=string`
  - **Response**: `[ { "restaurantId": "string", "name": "string", "rating": "float", "location": "string" } ]`

rder Placement

- **Endpoint**: `/api/v1/orders`
  - **Method**: POST

- **Request**: `{ "userId": "string", "restaurantId": "string", "items": [ { "itemId": "string", "quantity": "int" } ], "instructions": "string" }`
- **Response**: `{ "orderId": "string", "message": "Order placed successfully" }`

ayment Processing

- **Endpoint**: `/api/v1/orders/{orderId}/payment`
  - **Method**: POST
  - **Request**: `{ "paymentMethod": "string", "amount": "float" }`
  - **Response**: `{ "transactionId": "string", "message": "Payment successful" }`

rder Tracking

- **Endpoint**: `/api/v1/orders/{orderId}/status`
  - **Method**: GET
  - **Response**: `{ "orderId": "string", "status": "string", "estimatedDeliveryTime": "string" }`

eview and Rating System

- **Endpoint**: `/api/v1/restaurants/{restaurantId}/reviews`
  - **Method**: POST
  - **Request**: `{ "userId": "string", "rating": "int", "comment": "string" }`
  - **Response**: `{ "reviewId": "string", "message": "Review submitted successfully" }`

tabase Design

atabase Schema

- **User Table**
  - Fields: `user_id (PK)`, `email`, `password_hash`, `phone`, `created_at`
- **Restaurant Table**
  - Fields: `restaurant_id (PK)`, `name`, `location`, `cuisine_type`, `rating`
  - Relationships: One-to-many with Menu, Reviews
- **Order Table**
  - Fields: `order_id (PK)`, `user_id (FK)`, `restaurant_id (FK)`, `status`, `total_amount`, `created_at`
  - Relationships: Many-to-one with User, Restaurant
- **Payment Table**
  - Fields: `payment_id (PK)`, `order_id (FK)`, `amount`, `payment_method`, `transaction_id`, `status`
  - Relationships: One-to-one with Order
- **Review Table**
  - Fields: `review_id (PK)`, `user_id (FK)`, `restaurant_id (FK)`, `rating`, `comment`, `created_at`
  - Relationships: Many-to-one with User, Restaurant

ıtabase Choice and Rationale

- **Type**: Relational Database (e.g., PostgreSQL, MySQL)
- **Rationale**:
    - ACID Compliance: Ensures data consistency and integrity for transactions.
    - Structured Data: Well-defined relationships make a relational database suitable.
    - Complex Queries: Efficiently handles complex queries and joins.

ırformance Optimizations

- **Indexes**: Create indexes on `user_id`, `restaurant_id`, `order_id` for faster reads.
- **Partitioning**: Partition large tables like `Order` by date for improved performance.
- **Caching**: Use caching (e.g., Redis) for frequently accessed data to reduce load.
- **Connection Pooling**: Manage connections efficiently to reduce latency.
- **Read Replicas**: Distribute read traffic to enhance performance during peak times.

ɟh Level Architecture

ɣstem Design Overview

- **Client Applications**
    - Mobile App (iOS, Android), Web App
    - User interfaces for browsing restaurants, placing orders, and managing accounts.
- **API Gateway**
    - Single entry point for client requests, routing to backend services.
    - Handles authentication and request logging.
- **Backend Services**
    - User Service: Manages user profiles and authentication.
    - Restaurant Service: Handles listings and menu management.
    - Order Service: Manages order creation and tracking.
    - Payment Service: Processes payments.
    - Delivery Service: Assigns delivery personnel.
- **Database Layer**
    - Relational Database (e.g., PostgreSQL) for structured data storage.
- **Caching Layer**
    - Redis or Memcached for caching frequently accessed data.
- **Message Queue**
    - RabbitMQ or Kafka for asynchronous communication between services.
- **Load Balancer**
    - Distributes traffic across backend service instances.
- **Monitoring and Logging**
    - Tools like Prometheus and ELK Stack for performance monitoring and logging.

ıd-to-End Request Flow

- User opens the app and searches for restaurants.
- API Gateway routes the request to the Restaurant Service.
- Restaurant Service queries the database and caches results.
- API Gateway sends restaurant data back to the client.

- User places an order, routed through the API Gateway to the Order Service.
- Order Service validates the order and updates the database.
- Payment Service processes the payment.
- Delivery Service assigns a delivery person.
- Notification Service updates the user on order status.
- Order is delivered, and the user receives a final notification.

tailed Components Design

rder Service

- **Responsibilities**: Manages order creation, updates, and status tracking.
- **Design Considerations**:
    - **Scalability**: Microservices architecture for independent scaling.
    - **Data Consistency**: Implement transactions for atomic updates.
    - **Asynchronous Processing**: Use message queues for non-blocking updates.

ayment Service

- **Responsibilities**: Processes payments and manages transactions securely.
- **Design Considerations**:
    - **Security**: Encrypt sensitive data and comply with PCI DSS.
    - **Idempotency**: Ensure payment operations are idempotent.

elivery Service

- **Responsibilities**: Manages delivery assignments and optimizes routes.
- **Design Considerations**:
    - **Real-Time Tracking**: Utilize GPS for accurate tracking.
    - **Route Optimization**: Implement algorithms for efficient routing.

otification Service

- **Responsibilities**: Sends order status and promotional notifications.
- **Design Considerations**:
    - **Multi-Channel Support**: Enable SMS, email, and push notifications.
    - **Asynchronous Processing**: Decouple notifications using message queues.

PI Gateway

- **Responsibilities**: Central entry point for client requests.
- **Design Considerations**:
    - **Security**: Use OAuth2 for authentication.
    - **Load Balancing**: Distribute requests to prevent overload.

ade-off Discussion

ystem Design Choices

- **Monolithic vs. Microservices Architecture**
    - **Final Choice:** Microservices
    - **Justification:** Scalability and independent scaling of components like Order, Payment, and Delivery services are prioritized over monolithic simplicity.

- **Relational vs. NoSQL Database**
  - **Final Choice:** Relational Database
  - **Justification:** Structured data and complex queries necessitate a relational database, although NoSQL may be used for caching or analytics.
- **Synchronous vs. Asynchronous Communication**
  - **Final Choice:** Asynchronous Communication
  - **Justification:** Message queues enhance responsiveness and reliability by managing traffic spikes without blocking operations.
- **Multi-Channel Notifications**
  - **Final Choice:** Multi-Channel Notifications
  - **Justification:** Offers SMS, email, and push notifications to improve user engagement and ensure timely communication.
- **Third-Party vs. In-House Payment Integration**
  - **Final Choice:** Third-Party Payment Integration
  - **Justification:** Quick implementation and compliance allow focus on core functionalities while leveraging established payment providers.
- **Automated vs. Manual Delivery Assignment**
  - **Final Choice:** Automated Delivery Assignment
  - **Justification:** Optimizes routes and reduces errors, ensuring timely deliveries and enhancing customer satisfaction.

ilure Scenario Discussion

erver Failures

- **Potential Failure Points:** Backend services, database servers.
- **Strategies:**
  - Load Balancing: Distribute requests across multiple servers for redundancy.
  - Auto-Scaling: Adjust server instances based on traffic patterns.
  - Failover Mechanisms: Switch to backup servers during primary failures.

etwork Issues

- **Potential Failure Points:** Client-backend connectivity, internal service communication.
- **Strategies:**
  - CDN Usage: Cache static content to reduce latency.
  - Retry Logic: Implement retry mechanisms for transient failures.
  - Circuit Breaker Pattern: Prevent cascading failures by blocking requests to failing services.

atabase Failures

- **Potential Failure Points:** Database crashes, data corruption.
- **Strategies:**
  - Replication: Create data copies across servers for availability.
  - Backups: Regularly back up data to secure locations.

ervice Downtime

- **Potential Failure Points:** Maintenance, unexpected outages.

- **Strategies:**
    - Blue-Green Deployment: Minimize downtime during updates.
    - Graceful Degradation: Maintain partial functionality during outages.

ata Loss

- **Potential Failure Points:** Accidental deletions, hardware failures.
- **Strategies:**
    - Data Backups: Store backups in diverse locations.
    - Versioning: Recover from accidental deletions.

ıird-Party Service Failures

- **Potential Failure Points:** Payment gateways, mapping services.
- **Strategies:**
    - Redundancy: Use multiple services for alternatives.
    - Graceful Fallback: Queue payments for later processing.

isaster Recovery

- **Potential Failure Points:** Natural disasters, data center outages.
- **Strategies:**
    - Geographic Redundancy: Deploy services across multiple regions.
    - Disaster Recovery Plan: Regularly test recovery plans.