**HIGH LEVEL DESIGN DOCUMENT**

**UBER Eats**

**Prepared by** : Bhavana

**Table of Contents**

**1. General Overview**

i. User Interface

ii. Backend Components

iii. Database Schema

iv. API Endpoints

v. Infrastructure

vi. Integration

**2. Features**

i. User Authentication

ii. Filter Search

iii. Restaurant Management

iv. Payment Management

v. Delivery Tracking

**3. Architecture**

**4. System Design**

**5. Software and Hardware Requirements**

i. Technology Stack

ii. Deployment

iii. Testing

**6. Security Measures**

**Overview**

1. **User Interface (UI):**
   * **Homepage:** Displays featured restaurants, promotions, and search options.
   * **Search Interface**: Allows users to search for restaurants by location, cuisine, or name.
   * **Restaurant Listings**: Displays restaurants with relevant details like name, ratings, cuisine, and delivery options.
   * **Menu Interface**: Shows restaurant menus with food items, descriptions, and prices.
   * **Ordering Interface**: Allows users to add items to their cart, customize orders, and choose delivery or pickup options.
   * **Checkout Process:** Guides users through the payment process, including entering delivery details and selecting payment methods.
   * **User Profile**: Enables users to manage their account information, view order history, and track deliveries.
2. **Backend Components:**
   * **Authentication Service:** Handles user authentication and authorization.
   * **User Management Service:** Manages user accounts, profiles, and preferences.
   * **Restaurant Management Service:** Handles restaurant data, including listings, menus, and reviews.
   * **Order Management Service**: Manages the order lifecycle, including order placement, processing, and delivery tracking.
   * **Payment Service:** Integrates with payment gateways to process online transactions securely.
   * **Search Service:** Provides search functionality for finding restaurants based on various criteria.
   * **Notification Service:** Sends notifications to users regarding order updates, promotions, and other relevant information.
3. **Database Schema:**
   * **User Database**: Stores user information such as username, email, password hash, and preferences.
   * **Restaurant Database**: Stores restaurant details, including name, location, cuisine, menu items, and ratings.
   * **Order Database:** Tracks order information, including order ID, user ID, restaurant ID, items ordered, status, and delivery details.
   * **Review Database:** Stores user reviews and ratings for restaurants.
4. **API Endpoints:**
   * **Authentication Endpoints**: Handles user login, registration, and token generation.
   * **User Endpoints:** Provides CRUD operations for managing user accounts and profiles.
   * **Restaurant Endpoints:** Enables CRUD operations for managing restaurant listings, menus, and reviews.
   * **Order Endpoints:** Manages order placement, tracking, and history.
   * **Payment Endpoints:** Integrates with payment gateways to process transactions.
   * **Search Endpoints:** Facilitates restaurant search based on various criteria.
   * **Notification Endpoints:** Sends notifications to users via email, SMS, or push notifications.
5. **Infrastructure:**
   * **Cloud Hosting:** Deploys application components on scalable cloud infrastructure for reliability and performance.
   * **Load Balancing:** Distributes incoming traffic across multiple instances of application servers for optimal performance.
6. **Integration Points:**
   * Third-party APIs: Integrates with external services for features like geolocation, payment processing, and push notifications.
   * Payment Gateways: Implements payment gateways to securely process online transactions.
   * Geolocation Services: Utilizes geolocation services to determine user locations and provide accurate restaurant search results.

**Features**

**User Authentication**

* **Client Application**: The frontend application where users interact with the login and registration screens.
* **Backend Server**: The server-side application responsible for handling user authentication and registration requests.
* **Database:** Stores user credentials and other related information securely.

**1. User Registration:**

* When a user wants to register, they provide their details such as email, username, password, etc. on the client application.
* The client application sends a registration request to the backend server.
* The backend server receives the request, validates the provided information, and generates a unique user ID.
* It then hashes the password for security and stores the user details (including the hashed password) in the database.
* Once the registration is successful, the server sends a confirmation response to the client application.

**2. User Login:**

* When a user wants to log in, they provide their credentials (email/username and password) on the client application.
* The client application sends a login request to the backend server.
* The backend server receives the request and checks if the provided email/username exists in the database.
* If the user exists, it retrieves the stored hashed password from the database and compares it with the hashed password provided by the user during login.
* If the passwords match, the server generates a JWT (JSON Web Token) containing the user's ID and other relevant information.
* The server sends the JWT as a response to the client application, indicating successful login.
* The client application stores the JWT securely (e.g., in local storage or session storage) for subsequent authenticated requests.

**3. Security Considerations:**

* **Password Hashing:** Storing passwords as plaintext is a security risk. Therefore, passwords should be hashed using strong cryptographic hashing algorithms (e.g., bcrypt) before storing them in the database.
* **HTTPS:** All communications between the client application and the backend server should be encrypted using HTTPS to prevent eavesdropping and man-in-the-middle attacks.
* **JWT Authentication:** JWTs should be signed using a secure algorithm (e.g., HMAC-SHA256) and should include an expiration time to mitigate the risk of token theft and replay attacks.

**4. Scalability and Performance:**

* **Load Balancing:** Implementing load balancing techniques (e.g., round-robin, least connections) to distribute incoming requests evenly across multiple backend server instances.
* **Caching:** Implementing caching mechanisms (e.g., Redis) to cache frequently accessed data (e.g., user sessions) and improve performance.
* **Database Sharding:** As the user base grows, consider sharding the database to distribute data across multiple servers and handle increased load efficiently.

**5. Monitoring and Logging:**

* **Monitoring:** Implementing monitoring solutions (e.g., Prometheus, Grafana) to monitor server health, performance metrics, and user activity.
* **Logging:** Logging relevant events and errors to centralized logging systems (e.g., ELK stack) for troubleshooting and auditing purposes.

**Filter Search**

**1. Search by Location**:

Allow users to search for restaurants based on their current location or a specified address. Implement geocoding functionality to convert user inputs into geographic coordinates for accurate location-based searches.

**2. Filter by Cuisine Type**:

Enable users to filter restaurants by cuisine type (e.g., Italian, Chinese, Indian) to find establishments that offer their preferred cuisine.

**3. Refine by Price Range**:

Provide options for users to set price ranges to filter restaurants based on their budget preferences. Allow for flexible filtering, such as filtering by cheap, moderate, or expensive options.

**4. Sort by Rating and Popularity**:

Implement sorting options to arrange search results based on restaurant ratings, popularity, distance, or relevance. Allow users to customize sorting criteria to prioritize factors that are most important to them.

**Restaurant Management**

1. **Create/Update/Delete Restaurant Profile**:

Allow restaurant owners to create new profiles for their establishments with details such as name, address, cuisine type, contact information, opening hours, and menu items. Support operations for updating or deleting existing restaurant profiles as needed.

1. **Manage Menu Items**:

Provide functionalities for adding, editing, and removing menu items from restaurant profiles.Enable restaurant owners to specify details like item name, description, price, dietary information, and availability.

1. **Handle Reviews and Ratings**:

Allow users to submit reviews and ratings for restaurants based on their dining experiences. Implement mechanisms for displaying average ratings and reviews on restaurant profiles.

1. **Ordering and Delivery**:

Integrate online ordering and delivery functionalities to enable users to place food orders directly from restaurant profiles. Collaborate with delivery partners or establish an in-house delivery system for fulfilling orders.

1. **Promotions and Events**:

Allow restaurants to showcase special promotions, discounts, and events on their profiles to attract customers. Implement features for scheduling and managing promotional campaigns and events.

**Payment Management**

**1. Adding Payment Methods:**

Allow users to securely add and manage payment methods such as credit/debit cards, digital wallets, and bank accounts to their Zomato accounts.

**2. Making Payments:**

Enable users to make payments for food orders, restaurant reservations, delivery fees, etc., using their preferred payment methods. Implement secure checkout flows with real-time validation and error handling to ensure smooth payment processing.

**3. Transaction History:**

Provide users with access to their transaction history, including details such as transaction date, amount, payment method used, and transaction status (e.g., pending, completed, failed).

**4. Payment Notifications:**

Send email or push notifications to users to confirm successful payments, notify them of failed transactions, or request additional information for verification purposes.

**5. Refunds and Disputes:**

Implement procedures for handling refund requests, disputes, and chargebacks in accordance with payment gateway policies and regulations. Provide users with a mechanism to initiate refund requests and resolve payment-related issues promptly and transparently.

**Delivery Tracking**

**1. Order Status Updates:**

Implement mechanisms to update the status of orders in real-time as they progress through various stages of the delivery process (e.g., order placed, order accepted by restaurant, food preparation, out for delivery, delivered).Use WebSocket or server-sent events (SSE) to push status updates to the frontend in real-time, allowing users to track the progress of their orders without manual refreshes.

**2. Delivery Agent Assignment:**

Assign delivery agents to orders based on factors such as proximity to the restaurant, availability, workload, and delivery preferences. Implement algorithms or business rules to optimize delivery routes and minimize delivery times for multiple orders assigned to the same delivery agent.

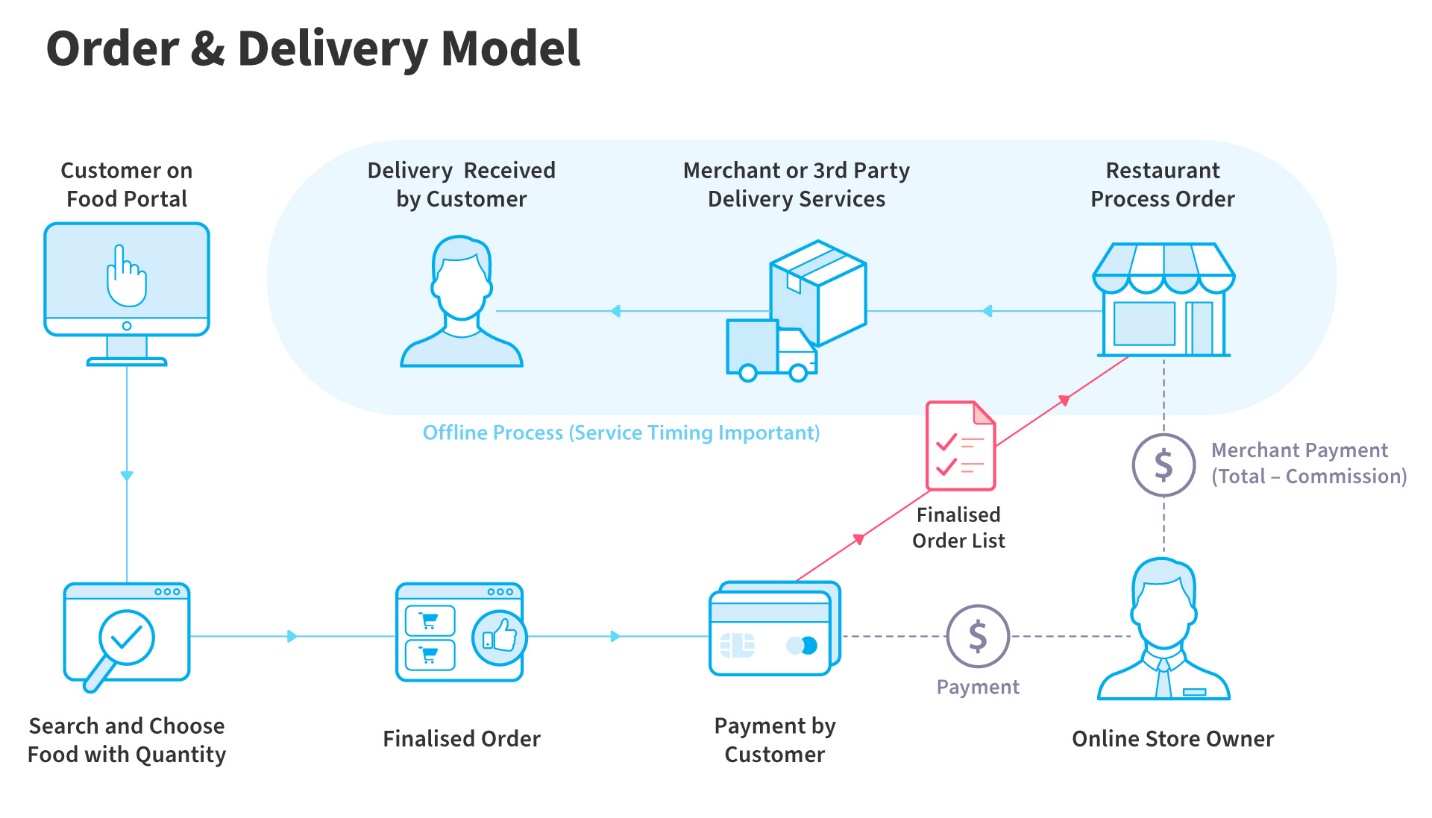
**3. Live Tracking Map:**

Provide users with a live tracking map interface that displays the real-time location of their delivery agent as they make their way to the delivery destination. Integrate with mapping and geolocation services (e.g., Google Maps API) to visualize delivery routes, estimate arrival times, and provide turn-by-turn navigation for delivery agents.

**4. Delivery Agent App:**

Develop a mobile application for delivery agents to manage their deliveries, view order details, navigate to delivery locations, and update order statuses (e.g., order picked up, enroute, delivered). Implement secure authentication and authorization mechanisms to ensure that only authorized delivery agents can access the app and perform delivery-related actions.

**Architecture**



**System Design**

The system design for the above features in the Zomato app involves designing scalable, reliable, and efficient components to handle user requests, process data, and provide a seamless experience. Here's an overview of the system design:

**1. Microservices Architecture:**

* Adopt a microservices architecture to decouple different functionalities of the system, such as user management, restaurant management, order processing, payment handling, and search functionality.
* Each microservice focuses on a specific domain and communicates with others via lightweight protocols like HTTP/REST or messaging queues.

**2. Authentication and Authorization:**

Implement a centralized authentication service using OAuth 2.0 or JWT (JSON Web Tokens) to handle user authentication and authorization across all microservices.

**3. User Management:**

* Develop a user management microservice responsible for user registration, login, profile management, and authentication.
* Utilize a scalable database to store user information securely, with features for user verification, password hashing, and access control.

**4. Restaurant Management:**

* Create a restaurant management microservice to handle CRUD operations for restaurant data, including creation, retrieval, update, and deletion of restaurant profiles, menus, reviews, and ratings.
* Implement a robust database schema optimized for querying restaurant information efficiently.

**5. Order Processing:**

* + Design an order processing microservice responsible for managing the order lifecycle, including order placement, tracking, fulfillment, and cancellation.
  + Integrate with third-party APIs or payment gateways to facilitate secure payment processing for orders.

**6. Search Functionality:**

* Develop a search microservice to enable users to discover restaurants based on various criteria such as location, cuisine type, ratings, and popularity.
* Implement indexing and search algorithms to optimize search queries for fast and accurate results.

**7. Notification Service:**

* Implement a notification microservice to send real-time notifications to users regarding order status updates, reservation confirmations, promotional offers, and other relevant events.
* Utilize push notification services for mobile devices and email or SMS gateways for web users.

**8. Infrastructure and Scalability:**

* Deploy the system on cloud infrastructure (e.g., AWS, Azure, Google Cloud) for scalability, high availability, and fault tolerance.
* Utilize containerization (e.g., Docker) and orchestration tools (e.g., Kubernetes) for managing microservices deployment and scaling.
* Implement auto-scaling policies to handle fluctuations in user traffic and ensure optimal resource utilization.

**Software and Hardware Requirements:**

**Technology Stack**

1. **Frontend Development:**
   * Framework: Angular or React for building responsive, interactive user interfaces.
   * Languages: HTML, CSS, JavaScript/TypeScript.
   * UI Libraries: Bootstrap or Material UI for styling and layout.
   * State Management: Redux or NgRx for managing application state (if using Angular).
   * HTTP Client: Axios or Angular Http Client for making HTTP requests to backend services.
2. **Backend Development:**
   * Framework: Spring Boot for building RESTful APIs and microservices.
   * Language: Java for backend logic implementation.
   * Database: PostgreSQL or MySQL for storing user data, restaurant details, orders, etc.
   * ORM: Spring Data JPA or Hibernate for object-relational mapping.
   * Authentication: Spring Security for handling user authentication and authorization.
   * REST Documentation: Swagger or Springfox for documenting and testing REST APIs.
3. **Microservices Architecture:**
   * Service Discovery: Netflix Eureka or Spring Cloud Netflix for service registration and discovery.
   * API Gateway: Spring Cloud Gateway or Zuul for routing and load balancing API requests.
   * Inter-Service Communication: Spring Cloud Feign or RestTemplate for communication between microservices.
   * Event Sourcing/Message Queues: Apache Kafka or RabbitMQ for asynchronous messaging and event-driven architecture.
4. **Database Management:**
   * Database: PostgreSQL or MySQL for relational data storage.
   * Data Access Layer: Spring Data JPA for interacting with the database.

**5. DevOps and Deployment:**

* + **Containerization**: Docker for containerizing microservices.
  + **Orchestration**: Kubernetes for container orchestration and management.
  + **CI/CD**: Jenkins or GitLab CI/CD for continuous integration and delivery pipelines.
  + **Monitoring and Logging**: Prometheus for monitoring metrics and ELK Stack (Elasticsearch, Logstash, Kibana) for centralized logging.

**Security**:

* + **JWT**: JSON Web Tokens for stateless authentication and authorization.
  + **OAuth 2.0**: For delegated authorization and third-party authentication.
  + **Encryption**: SSL/TLS for securing communication channels and sensitive data encryption.

**Testing**:

* + **Unit Testing**: JUnit and Mockito for unit testing backend services.
  + **Integration Testing**: Spring Boot Test for testing REST APIs and microservices.
  + **End-to-End Testing**: Selenium or Cypress for automated UI testing.

**Security Measures**

1. **User Authentication and Authorization:**
   * Use industry-standard authentication mechanisms such as JWT (JSON Web Tokens) or OAuth 2.0 for user authentication.
   * Implement secure password storage by hashing passwords using strong hashing algorithms like bcrypt.
   * Enforce password policies such as minimum length, complexity, and expiration to enhance password security.
   * Implement role-based access control (RBAC) to restrict access to certain features based on user roles (e.g., customer, restaurant owner, admin).
2. **HTTPS Encryption:**
   * Ensure that all communication between the client-side application and the server-side APIs is encrypted using HTTPS.
   * Use SSL/TLS certificates to establish secure connections and prevent man-in-the-middle attacks.
3. **Input Validation and Sanitization:**
   * Validate and sanitize all user inputs on the client-side and server-side to prevent injection attacks such as XSS (Cross-Site Scripting) and SQL injection.
   * Use input validation libraries or frameworks to enforce input validation rules and reject malformed or malicious inputs.
4. **API Security:**
   * Secure APIs by implementing authentication and authorization mechanisms to restrict access to authorized users and prevent unauthorized access to sensitive data.
   * Use API keys, OAuth tokens, or JWT tokens for API authentication and enforce API rate limiting and throttling to mitigate abuse.
5. **Data Encryption:**
   * Encrypt sensitive data such as user credentials, payment information, and personal data at rest using strong encryption algorithms.
   * Use encryption libraries or frameworks to handle data encryption and decryption securely.
6. **Security Auditing and Logging:**
   * Implement comprehensive logging mechanisms to record security-related events and monitor system activities for suspicious behavior.
   * Conduct regular security audits and vulnerability assessments to identify and address potential security vulnerabilities proactively.