LOW LEVEL DESIGN DOCUMENT FOR SWIGGY:

Swiggy is an Indian online food ordering and delivery platform. Founded in 2014, Swiggy is headquartered in Bangalore and operates in more than 500 Indian cities as of September 2021.

Swiggy, being a popular food delivery platform, offers various features to its users, restaurants, and delivery partners. Here is the Low-Level design key features for Swiggy:

FEATURE 1: FOR CUSTOMER (LOGIN – PAGE):

COMPONENTS:

Html form: User interface elements for capturing username/email and password.

JavaScript: Client-side scripting for form validation and interaction.

AJAX: Asynchronous requests to communicate with the backend authentication service.

API Endpoint: Exposes a /login endpoint to handle authentication requests.

Database Access: Interacts with the user database to verify credentials.

Token Generation: Generates and returns an authentication token upon successful login.

Interaction Flow:

User Input: Customers enter their username/email and password into the login form.

Client-side Validation: JavaScript validates input fields for completeness, format, and length.

AJAX Request: Upon form submission, an AJAX request is sent to the backend authentication endpoint.

Receive Request: Listens for incoming login requests at the /login endpoint.

Extract Credentials: Extracts username/email and password from the request payload.

Database Query: Queries the user database to retrieve the user's stored credentials.

Password Verification: Compares the hashed password from the database with the provided password.

Token Generation: If credentials match, generates an authentication token (JWT) containing user details.

Response: Sends the token in the response payload along with a success status code (200).

SECURITY MEASURES:

Encryption: Ensure that all communication between the frontend and backend is encrypted using HTTPS.

Password Hashing: Store passwords securely using a strong hashing algorithm

Brute Force Protection: Implement rate-limiting to prevent brute force attacks on the login endpoint.

Token Expiry: Set a short expiry time for authentication tokens to limit their lifespan and reduce the risk of misuse.

Cross-Site Scripting (XSS) Protection: Sanitize user input to prevent XSS attacks on the frontend.

SESSION MANAGEMENT:

Token Storage: Frontend stores the authentication token securely (e.g., local storage, cookies).

Token Expiry: Backend sets a short expiry time for tokens and handles token refresh for long-lived sessions.

Logout Functionality: Provides users with the option to log out, clearing their session token.

ERROR HANDLING:

Invalid Credentials: Return a 401 Unauthorized status code for invalid login attempts.

Server Errors: Handle server errors gracefully and return appropriate error responses (e.g., 5xx status codes).

LOGGING AND MONITORING:

Logging: Log login attempts, including IP addresses and timestamps, for security auditing purposes.

Monitoring: Implement monitoring tools to track login success rates, error rates, and system performance.

TESTING:

Unit Testing: Write unit tests to ensure the correctness of password hashing, token generation, and database interactions.

Integration Testing: Test the login functionality end-to-end, including frontend-backend interactions and error handling.

This low-level design outlines the detailed implementation of Swiggy' s customer login page, covering both frontend and backend components, security measures, session management, error handling, logging, monitoring, and testing considerations.

FEATURE 2: RESTAURANTS PAGE

COMPONENTS:

HTML/CSS/JavaScript: Frontend components for displaying restaurant details, menu items, and user interactions.

API Requests: AJAX or Fetch API for fetching restaurant data from backend services.

UI Frameworks/Libraries: Utilize frameworks like React, Vue.js, or Angular for building interactive and responsive user interfaces.

Interaction Flow:

Fetch Restaurant Data: Upon loading the page, frontend sends a request to the backend API to fetch restaurant details based on the selected location or search query.

Display Restaurant Information: Render restaurant name, address, contact information, opening hours, delivery options, ratings, and reviews.

Display Menu: Render the restaurant's menu with categories, items, descriptions, prices, and images.

Handle User Interactions: Allow users to customize their orders, add items to the cart, specify delivery preferences, and proceed to checkout.

Handle Favorites and History: Implement functionality for users to mark favorite restaurants and view their order history.

BACKEND SERVICES:

RESTful API: Exposes endpoints for fetching restaurant details, menu items, ratings, and reviews.

Database Access Layer: Interacts with the database to retrieve restaurant data based on user requests.

Business Logic Layer: Implements business rules for filtering and sorting restaurant listings, handling menu customization, and managing user interactions.

INTERACTION FLOW:

Receive Request: Backend API receives requests from the frontend for restaurant data.

Query Database: Database access layer executes queries to retrieve restaurant details, menu items, ratings, and reviews based on the request parameters.

Process Data: Business logic layer processes the retrieved data, applies filters, sorts results, and formats the response.

Send Response: Backend API sends the formatted data as a response to the frontend for display.

DATABASE SCHEMA:

Restaurants: Stores information about registered restaurants, including name, address, contact details, opening hours, and delivery options.

Menu Items: Contains details of menu items offered by each restaurant, including name, description, price, category, and image URL.

Ratings: Stores user ratings and reviews for each restaurant, along with timestamps and user identifiers.

SECURITY MEASURES:

Authentication and Authorization: Ensure that only authenticated users can access restaurant data and place orders.

Input Validation: Validate input parameters to prevent injection attacks and ensure data integrity.

Rate Limiting: Implement rate limiting to prevent abuse and protect against denial-of-service attacks.

Data Encryption: Encrypt sensitive data such as user credentials, session tokens, and payment information to protect against unauthorized access.

Error Handling:

Handle Invalid Requests: Return appropriate error responses (e.g., 404 Not Found, 400 Bad Request) for invalid requests or missing data.

Logging: Log errors and exceptions for debugging and troubleshooting purposes.

Caching:

Cache Restaurant Data: Implement caching mechanisms to cache frequently accessed restaurant data and improve performance.

Cache Invalidation: Set expiration times for cached data and implement cache invalidation strategies to ensure data freshness.

TESTING:

Unit Testing: Write unit tests to ensure the correctness of database queries, API endpoints, and business logic.

Integration Testing: Test the integration between frontend and backend components, including API requests and responses.

MONITORING AND LOGGING:

Monitoring Tools: Utilize monitoring tools to track system performance, detect anomalies, and monitor resource usage.

Logging: Log relevant events, errors, and user interactions for auditing and analysis purposes.

This low-level design provides a detailed implementation plan for building Swiggy's restaurant page, covering frontend components, backend services, database schema, security measures, error handling, caching, testing, and monitoring considerations.

FEATURE 3: OFFER PAGE

COMPONENTS:

HTML/CSS/JavaScript: Frontend components for rendering offer listings, details, and user interactions.

API Requests: Utilize AJAX or Fetch API to communicate with backend services and fetch offer data.

UI Frameworks/Libraries: Optionally, use frameworks like React, Vue.js, or Angular for building dynamic and responsive user interfaces.

INTERACTION FLOW:

Fetch Offer Data: Upon page load, frontend sends a request to the backend API to fetch available offers.

Render Offer Listings: Display offer listings, including offer title, description, discount percentage, validity period, and associated restaurants or partners.

Handle User Interactions: Allow users to explore offers, filter by category or location, and view offer details.

Redemption Flow: Provide options for users to redeem offers, apply coupon codes, and activate deals during checkout.

Receive Request: Backend API receives requests from the frontend for offer data or redemption actions.

Query Database: Database access layer executes queries to retrieve offer details, including title, description, discount, validity period, and associated restaurants/partners.

Process Data: Business logic layer filters and formats offer data, applying any business rules or conditions (e.g., offer eligibility, redemption limits).

Send Response: Backend API sends the formatted offer data or redemption status as a response to the frontend.

BACKEND SERVICES:

RESTful API: Exposes endpoints for fetching offer data, applying discounts, and managing offer redemption.

Database Access Layer: Interacts with the database to retrieve offer information based on user requests.

Business Logic Layer: Implements business rules for filtering offers, processing redemption requests, and tracking offer usage.

DATABASE SCHEMA:

Offers: Stores information about available offers, including offer ID, title, description, discount percentage, validity period, and redemption details.

Redemptions: Tracks offer redemptions by users, including user ID, offer ID, redemption date, and status (redeemed, expired, etc.).

SECURITY MEASURES:

Authentication and Authorization: Ensure that only authenticated users can access offer-related functionalities and redeem offers.

Data Encryption: Encrypt sensitive offer data, such as coupon codes and redemption tokens, to prevent interception and misuse.

Input Validation: Validate user input and API requests to prevent injection attacks, XSS vulnerabilities, and data manipulation attempts.

Secure Communications: Implement HTTPS protocol to encrypt data in transit and protect against eavesdropping and tampering.

ERROR HANDLING:

Handle Invalid Requests: Return appropriate error responses (e.g., 404 Not Found, 400 Bad Request) for invalid requests or missing data.

Logging: Log errors and exceptions for debugging and troubleshooting purposes.

CACHING:

Cache Offer Data: Implement caching mechanisms to cache frequently accessed offer data and improve performance.

Cache Invalidation: Set expiration times for cached data and implement cache invalidation strategies to ensure data freshness.

TESTING:

Unit Testing: Write unit tests to ensure the correctness of database queries, API endpoints, and business logic.

Integration Testing: Test the integration between frontend and backend components, including API requests and responses.

MONITORING AND LOGGING:

Monitoring Tools: Utilize monitoring tools to track system performance, detect anomalies, and monitor resource usage.

Logging: Log relevant events, errors, and user interactions for auditing and analysis purposes.

This low-level design outlines the detailed implementation plan for building Swiggy's offer page, covering frontend and backend components, database schema, security measures, error handling, caching, testing, and monitoring considerations.

FEATURE 4: ORDER PAGE

COMPONENTS:

HTML/CSS/JavaScript: Frontend components for displaying order summary, customization options, delivery preferences, and payment processing.

API Requests: AJAX or Fetch API for interacting with backend services to fetch order details, process payments, and track order status.

UI Frameworks/Libraries: Utilize frameworks like React, Vue.js, or Angular for building interactive and responsive user interfaces.

INTERACTION FLOW:

Display Order Summary: Render a summary of selected items, quantities, prices, and total order value for review before checkout.

Customization Options: Allow users to customize their orders by selecting options such as size, toppings, sides, special instructions, and preferences.

Delivery Preferences: Provide input fields for users to specify delivery address, time preferences, contact details, and any specific delivery instructions or notes.

Payment Processing: Render payment options and capture user payment details securely using payment gateway APIs.

Order Confirmation: Display an order confirmation message with details such as order ID, estimated delivery time, and payment confirmation upon successful checkout.

Receive Order Request: Backend API receives requests from the frontend to place an order, including order details, customization options, and delivery preferences.

Process Payment: Payment processing service verifies user payment details, charges the appropriate amount, and confirms payment authorization with the payment gateway.

Update Order Status: Order management service updates the order status in the database (e.g., from "Pending" to "Processing") and sends notifications to users and delivery partners.

Dispatch Order: Coordinates with delivery partners to dispatch the order for delivery to the specified address within the requested time frame.

Track Order Status: Provides APIs for frontend to fetch real-time updates on order status, including confirmation, preparation, dispatch, and delivery progress.

BACKEND SERVICES:

RESTful API: Exposes endpoints for handling order placement, payment processing, order tracking, and communication with users and delivery partners.

Order Management Service: Manages the order lifecycle, including order processing, payment handling, and delivery coordination.

Database Access Layer: Interacts with the database to retrieve and store order details, user information, payment records, and order status updates.

DATABASE SCHEMA:

Orders: Stores information about user orders, including order ID, user ID, order details, delivery address, payment status, and order status.

Users: Contains user information, including user ID, name, contact details, and delivery addresses.

Payments: Records payment transactions, including payment ID, order ID, payment amount, payment status, and timestamp.

SECURITY MEASURES:

Authentication and Authorization: Ensures that only authenticated users can access and place orders, enforcing user authentication and authorization mechanisms.

Data Encryption: Encrypts sensitive order data, such as payment information and delivery addresses, to protect against unauthorized access and data breaches.

Payment Security: Integrates with secure payment gateways and follows industry-standard security protocols (e.g., PCI DSS compliance) to ensure the security of payment transactions and protect users' financial information.

Input Validation: Validates user input and API requests to prevent injection attacks, XSS vulnerabilities, and data manipulation attempts, ensuring data integrity and system security.

ERROR HANDLING:

Handle Invalid Requests: Return appropriate error responses (e.g., 404 Not Found, 400 Bad Request) for invalid requests or missing data.

Logging: Log errors and exceptions for debugging and troubleshooting purposes.

Testing:

Unit Testing: Write unit tests to ensure the correctness of database queries, API endpoints, and business logic.

Integration Testing: Test the integration between frontend and backend components, including API requests and responses.

Monitoring and Logging:

Monitoring Tools: Utilize monitoring tools to track system performance, detect anomalies, and monitor resource usage.

Logging: Log relevant events, errors, and user interactions for auditing and analysis purposes.

This low-level design provides a detailed implementation plan for building Swiggy's order page, covering frontend components, backend services, database schema, security measures, error handling, testing, and monitoring considerations.

FEATURE 5: SUPPORT PAGE

COMPONENTS:

HTML/CSS/JavaScript: Frontend components for displaying support options, FAQs, contact forms, and chat support.

API Requests: AJAX or Fetch API for interacting with backend services to fetch FAQs, submit support requests, and initiate chat sessions.

UI Frameworks/Libraries: Utilize frameworks like React, Vue.js, or Angular for building interactive and responsive user interfaces.

INTERACTION FLOW:

Display Support Options: Render various support options such as FAQs, contact forms, chat support, and helpline numbers.

FAQs Section: Display frequently asked questions and their answers to help users find solutions to common issues.

Contact Form: Provide a form for users to submit support requests, including their name, email, subject, and description of the issue.

Chat Support: Implement a chat support widget to enable real-time communication between users and support agents for immediate assistance.

Receive Support Request: Backend API receives support requests submitted by users via the contact form, containing details such as user information, issue description, and request type.

Create Support Ticket: Support ticket management service creates a new support ticket in the database, assigning it to an available support agent for resolution.

Fetch FAQs: Backend API retrieves frequently asked questions from the database and sends them to the frontend for display.

Initiate Chat Session: When a user requests chat support, backend API initiates a chat session, assigns a support agent, and establishes a WebSocket connection for real-time communication.

Handle Chat Messages: Chat support service manages incoming messages from users and support agents, routes messages to the appropriate recipients, and updates chat history in the database.

Resolve Support Ticket: Upon resolution of a support ticket, support ticket management service updates the ticket status in the database and sends a notification to the user.

BACKEND SERVICES:

RESTful API: Exposes endpoints for handling support requests, fetching FAQs, and initiating chat sessions.

Support Ticket Management: Manages support tickets submitted by users, including ticket creation, assignment to support agents, and resolution tracking.

Chat Support Service: Implements real-time chat functionality, including message exchange between users and support agents.

DATABASE SCHEMA:

SupportTickets: Stores information about support tickets, including ticket ID, user ID, issue description, status, and resolution details.

FAQs: Contains frequently asked questions and their corresponding answers.

ChatMessages: Records chat messages exchanged between users and support agents, including message ID, sender ID, recipient ID, timestamp, and message content.

<u>SECURITY MEASURES:</u>

Authentication and Authorization: Ensures that only authenticated users can access support features and submit support requests.

Data Encryption: Encrypts sensitive user data, such as contact information and support ticket details, to protect against unauthorized access and data breaches.

Input Validation: Validates user input and API requests to prevent injection attacks, XSS vulnerabilities, and data manipulation attempts, ensuring data integrity and system security.

<u>ERROR HANDLING:</u>

Logging: Log errors and exceptions for debugging and troubleshooting purposes.

<u>TESTING:</u>

Unit Testing: Write unit tests to ensure the correctness of database queries, API endpoints, and business logic.

Integration Testing: Test the integration between frontend and backend components, including API requests and responses.

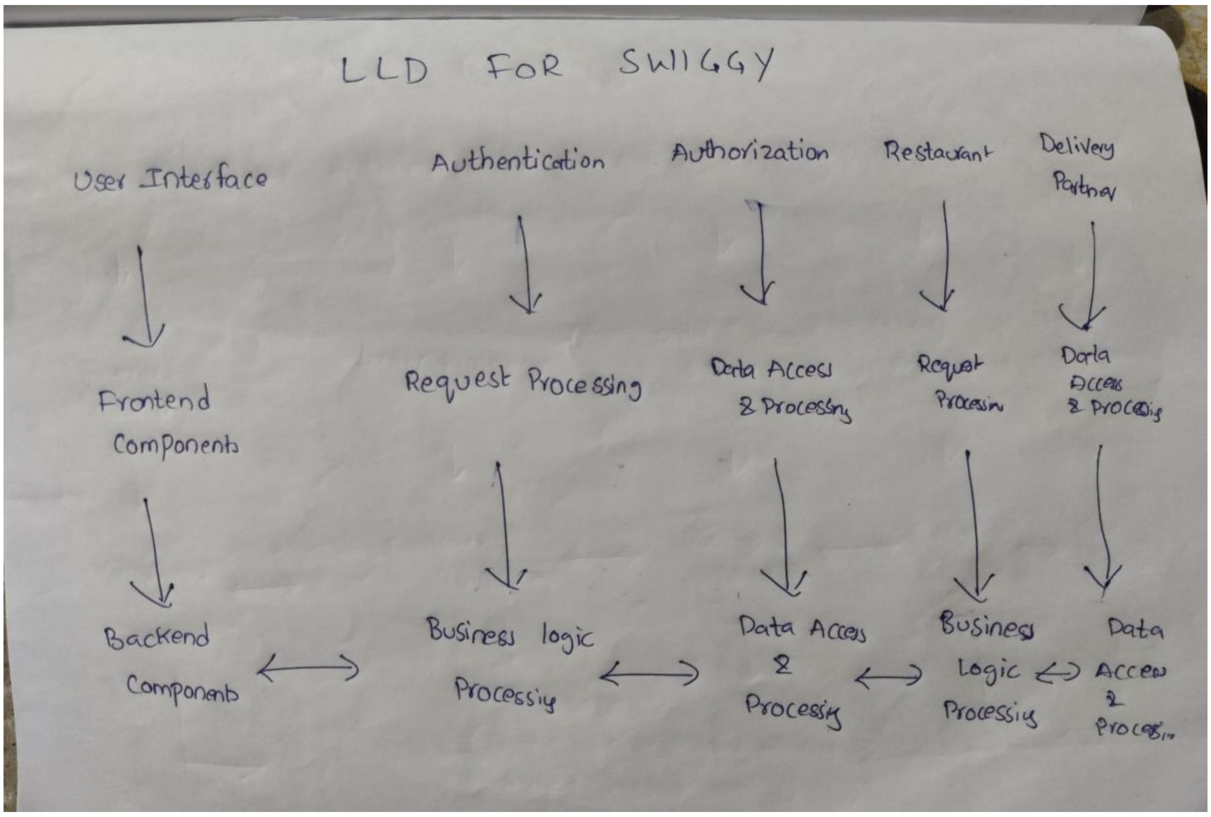
<u>MONITORING AND LOGGING:</u>

Monitoring Tools: Utilize monitoring tools to track system performance, detect anomalies, and monitor resource usage.

Logging: Log relevant events, errors, and user interactions for auditing and analysis purposes.

This low-level design provides a detailed implementation plan for building Swiggy's support page, covering frontend components, backend services, database schema, security measures, error handling, testing, and monitoring considerations.

ARCHITECTURE OVERVIEW OF HLD FOR SWIGGY


LLD FOR SWIGGY
User Interface
Authentication
Authorization
Restaurant
Delivery Partner
Frontend Components
Request Processing
Data Access & Processing
Request Processing
Data Access & Processing
Backend Components
Business logic Processing
Data Access & Processing
Business Logic Processing
Data Access & Processing