System Design Document

Application Name: Foodie.com

Version: v1.0

Date: 27-04-2024

Author(s): Aditya Goyal

Contents

Introduction

Purpose

The purpose of this system design document is to outline the architecture, design decisions, and technical considerations for the development of “Foodie.com". This document aims to provide a comprehensive understanding of the system's components, functionality, and implementation details to guide the development team throughout the project lifecycle.

Scope

The scope of this document encompasses the design and implementation of a mobile and web-based food delivery platform that enables users to browse restaurants, place orders, track delivery status, and make payments online. The system will include features such as restaurant management, order processing, delivery management, and integration with payment gateways.

Audience

This document is intended for various stakeholders involved in the development, deployment, and maintenance of the food delivery application. The primary audience includes:

- **Development Team:** Developers, architects, and engineers responsible for building and implementing the system.
- **Product Managers:** Individuals overseeing the product development and responsible for defining feature requirements and priorities.
- **Quality Assurance Team:** Testers and quality assurance engineers responsible for validating the functionality and performance of the application.
- **Project Stakeholders:** Business owners, investors, and other stakeholders interested in understanding the technical aspects and capabilities of “Foodie.com”.

System Overview

Description

“Foodie.com” is a comprehensive platform designed to connect users with a wide range of restaurants and facilitate the ordering and delivery of food items. Through intuitive mobile and web interfaces, users can browse menus, place orders, and track their delivery in real-time. The application will support multiple features such as user authentication, restaurant management, order processing, delivery coordination, and secure payment processing.

Goals

The primary goals and objectives of “Foodie.com” are as follows:

- **Enhanced User Experience:** Provide a seamless and intuitive user experience across mobile and web platforms, allowing users to effortlessly discover restaurants, browse menus, place orders, and track deliveries.
- **Restaurant Partner Integration:** Enable restaurants to easily onboard onto the platform, manage their menus, and receive and fulfill orders efficiently.
- **Efficient Order Processing:** Streamline the order processing workflow to ensure timely and accurate order fulfillment, including real-time notifications and status updates for users and delivery partners.
- **Delivery Management:** Facilitate efficient coordination between delivery partners and users, optimizing delivery routes and providing real-time tracking.
- **Secure Payments:** Implement robust payment processing mechanisms to securely handle online transactions, supporting digital wallet for payments while ensuring compliance with industry standards and regulations.
- **Scalability and Performance:** Design the system to scale seamlessly to accommodate growing user traffic and transaction volumes, while maintaining high performance and reliability under peak load conditions.

Assumptions

Following assumptions have been made while writing this document:

- A foodie can order food from the restaurants within certain radius, let’s assume 8 kilometers.
- A foodie can not order items from different restaurants in one order.

Actors

- Foodies
- Partner Restaurants (Referred as “Merchants” in this document)
- Delivery Partners (Referred as “Pilots” in this document)

Functional Requirements

Foodies should be able to:

- Search for a restaurant by its name
- Search for a dish they would like to include in their meal
- Create a cart containing the items they would love to eat
- Place an order and pay for it via an integrated digital wallet
- Receive the status update on the order in the application
- Track the real time location of delivery partner
- Cancel the order

Merchant should be able to:

- Onboard their restaurants and create menu with the items they would like to serve

- Receive notifications when they receive an orders from the foodies
- Accept or reject the orders
- Receive notifications when pilots are assigned to deliver the orders
- Receive notifications when pilots successfully deliver the orders

Pilots should be able to:

- Receive notifications when there are orders for delivery nearby
- Receive notifications when the orders are ready to be picked up
- Accept the orders they wish to deliver
- Receive notifications when the orders are delivered

Non-functional Requirements

- Foodies should be able to search a restaurant or dish lightening fast
- When a new merchant is on-boarded or a menu item added by an existing merchant, it should be available to search within no time. However, minor latency is acceptable in this case as different services will be communicating with each other to reflect the data and make it available for search
- When an order is placed, the foodie, merchant and the pilot should be notified consistently
- Services should be highly available and scalable because the business revenue is directly dependent on the availability of the services
- Services should be fault tolerant and able to handle traffic during peak load time

Out Of Scope

- Customer service
- Contacting the pilot

Architecture

High Level Details

- As there will be many services playing different roles and communicating with each other, the system could be designed using a micro-services architecture.
- The overall design of the system will be based on heavy usage of messaging queues to setup communication protocols between different services.
- Different services will be using different type of data storage and retrieval mechanisms based on the workflows and supported operations. For example, the merchant service will be supporting a huge number of read operations as compared to write operations while the order service will be supporting a huge number of write operations as compared to read operations.
- Each service will be interacting with its own database (database-per-service paradigm) and a number of messaging queues to communicate with other services.
- One service will not be able to connect with other service's database directly.

- The co-ordinates of the foodie's address and merchant's address will be stored in the system because the search service will be filtering the restaurants based on the distance from the foodie's address.
- At the moment, only payment service will be communicating with an external system via payment gateway to handle the payments for the orders.
- Caching mechanisms will be used to make search operations faster e.g. searching a restaurant or dish.

Components

- **User Interface:** An actor can access the user interfaces via mobile app and web browsers depending on its role in the system. Initially, there will be three versions of user interface, one for each actor in the system. Each interface will communicate with a dedicated service to perform a set of operations to fulfill the user journey e.g. displaying a list of restaurant and their menu, searching for a restaurant or dish will be communicating with Search Ecosystem.
- **Search Ecosystem:** The journey of foodies start with searching for a restaurant or dish they'd love to include in their meal. This component of the system will be responsible to present different options available to select to the foodies.
    - It will deal with millions (or maybe more) of GET requests with filters and sorting options. Hence, it is going to be a read-heavy system.
    - We can use Elasticsearch or Apache Solr to execute these read operations because these technologies are open source and provide efficient ways to search, filter and sort the data stored in a cluster. Since merchants will be filtered based on the distance from the foodie, we need spatial search support which is provided by both of these technologies.
    - It needs to have a queue in place to process asynchronous updates to the search cluster. An event will be posted whenever a merchant is created/updated/removed.
    - This component will have an data indexer to consume these update events. The indexer will index appropriate data in the search cluster.
- **Order Service:** The order journey begins when a foodie has selected the menu items and added those int the cart.
    - This component will be responsible for managing items selection, shopping cart and order placement.
    - It will interact with payment service to process the payment made for each order.
    - The payment service will talk to an external gateway to process the payment and notify the status of the payment to the order service.
    - The orders and payments data will be stored in the orders database which will be a relational database like PostgreSQL.
    - The data stored in the order database will be used to generate order history of a foodie and payment invoices.
- **Order Fulfillment Service:** This service will act as a bridge between a foodie and the order to send and receive updates.
    - It will be notified when a merchant accepts/rejects an order.
    - It will notify the foodie when a merchant accepts/rejects an order.
    - It will notify a pilot when an order is ready to be picked up.
    - It will update the foodie about the status of an order.

    - All of this communication will go through messaging queue.
- **Account Management Services:** There will be three account management services supporting the system. One to manage foodie accounts, one to manage merchant accounts and one to manage pilot accounts.
    - Foodies can update their contact information, add one or more addresses, link digital wallet in their account. They can view their order history in their profile as well.
    - Merchants can update their contact information, add/remove/update items, update price of items in their menu, upload images of menu items. They can view the past orders they have fulfilled.
    - Pilots can manage their contact information and the area they will be delivering in through their profiles.
- **Pilot Service:** This component will be responsible for the following:
    - Manage nearby orders to be picked up in a list.
    - Accept/Reject a delivery.
    - View history of the orders personally delivered.
    - View contact information of the foodie from the order in case pilot needs to contact foodie while delivering the order.
- **Notification Service:** This component will be responsible for delivering the notifications to different actors in the system when required. There can be different ways to send notifications e.g. text SMS, emails, push notifications. Actors can choose to allow all types of notification or a few as per their convenience.
    - A foodie will receive notifications about the status of the order e.g. order placed, picked up, out for delivery, delivered.
    - A merchant will receive notifications about the order placed, pilot assigned, picked up, delivered.
    - A pilot will receive notifications when an order is waiting to be accepted for delivery, added in the queue of orders to be picked up.

External Systems

Initially, there is only one external system playing a crucial role in the system i.e. payment gateway. As the foodies are allowed to pay via a digital wallet, this wallet will have to be linked to the foodie's account and contacted for updating the wallet balance via the payment gateway. The payment gateway is responsible for sending the payment status to the order service. A payment can result in failed status in case of insufficient money in the wallet or if there are any network issues. The foodie should be able to make the payment for the same order again in case it has failed. There is no limit on number of times a foodie can retry to make the payment until it is successful. The payment related data passed by the gateway to the order service will be stored in the orders database which is a relational database to maintain ACID transactions.

Database Model

1. Users Table:

- **user_id**: Primary key

- **username**: Unique username for each user
- **email**: Email address of the user
- **password**: Encrypted password for authentication
- **first_name**: First name of the user
- **last_name**: Last name of the user
- **phone_number**: Phone number of the user
- **address**: Address of the user
- **created_at**: Timestamp indicating when the user account was created

2. Restaurants Table:

- **restaurant_id**: Primary key
- **name**: Name of the restaurant
- **description**: Description of the restaurant
- **address**: Address of the restaurant
- **phone_number**: Phone number of the restaurant
- **created_at**: Timestamp indicating when the restaurant was added to the platform

3. Menu Items Table:

- **item_id**: Primary key
- **restaurant_id**: Foreign key referencing the restaurant that offers the item
- **name**: Name of the menu item
- **description**: Description of the menu item
- **price**: Price of the menu item
- **category**: Category of the menu item (e.g., appetizer, main course, dessert)
- **created_at**: Timestamp indicating when the menu item was added

4. Orders Table:

- **order_id**: Primary key
- **user_id**: Foreign key referencing the user who placed the order
- **restaurant_id**: Foreign key referencing the restaurant from which the order was placed
- **total_amount**: Total amount of the order
- **status**: Status of the order (e.g., pending, confirmed, delivered)
- **created_at**: Timestamp indicating when the order was placed

5. Order Items Table (Many-to-Many Relationship between Orders and Menu Items):

- **order_item_id**: Primary key
- **order_id**: Foreign key referencing the order to which the item belongs
- **item_id**: Foreign key referencing the menu item that was ordered
- **quantity**: Quantity of the menu item ordered
- **subtotal**: Subtotal for the quantity of the menu item ordered

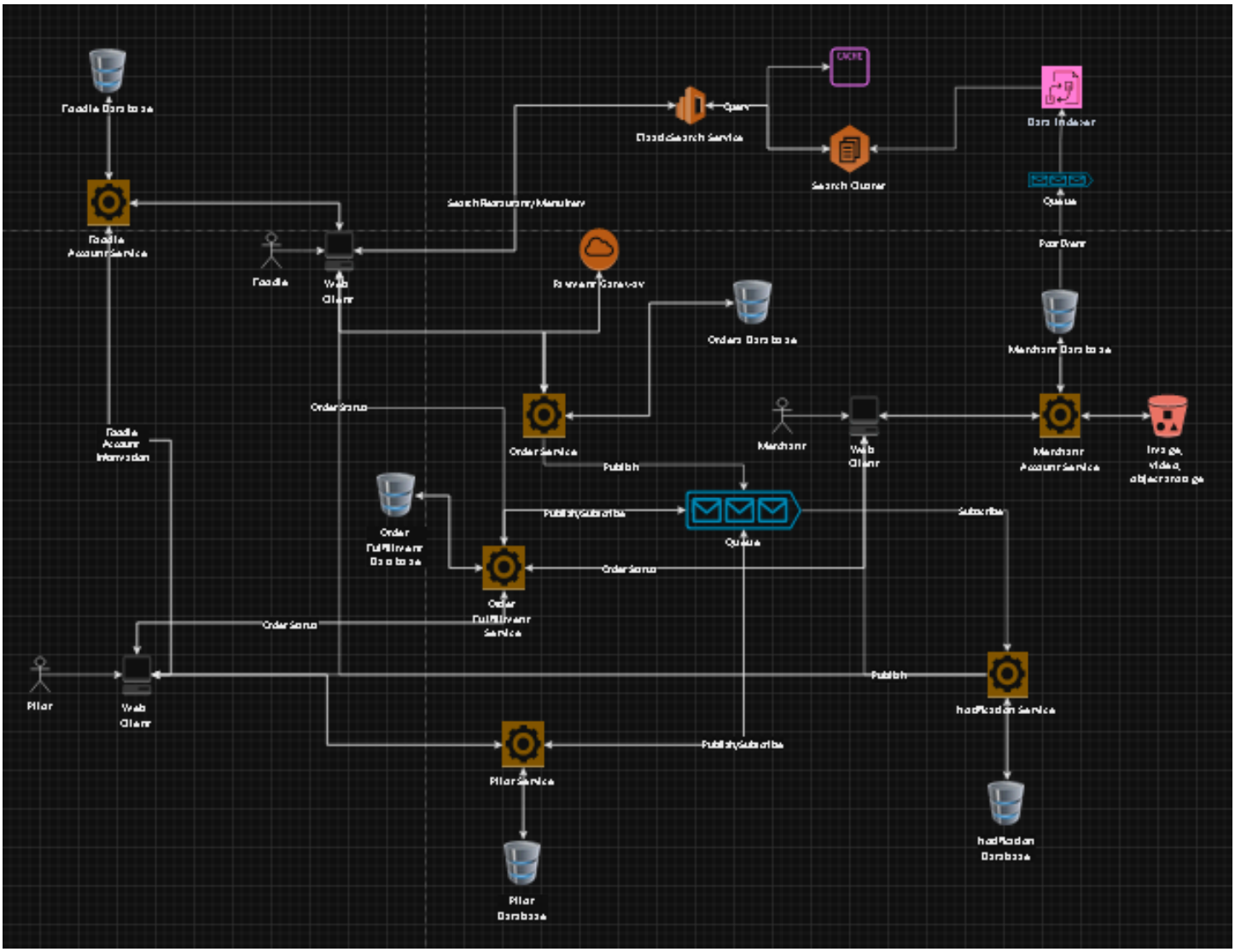
6. Delivery Tracking Table:

- **delivery_id**: Primary key
- **order_id**: Foreign key referencing the order being delivered

- **driver_id**: Foreign key referencing the driver assigned to deliver the order
- **status**: Status of the delivery (e.g., on the way, delivered)
- **estimated_delivery_time**: Estimated time of delivery
- **actual_delivery_time**: Actual time of delivery
- **delivery_notes**: Additional notes or instructions for the delivery

Data Flow Diagram

For more clarity and zoom options, please refer to the other attachment.



Technologies Used

Programming Languages

- **Back-end Development**:
  - **Language**: Node.js or Python are popular choices for their ease of use, extensive libraries, and suitability for building RESTful APIs.
  - **Framework**: Express.js for Node.js or Flask/Django for Python can be used to quickly develop robust back-end APIs.
- **Front-end Development**:

    - **Language**: JavaScript (with HTML/CSS) for building interactive and responsive user interfaces.
    - **Framework**: React.js or Vue.js for building modern single-page applications (SPAs) with component-based architecture and efficient state management.
- **Mobile App Development**:
    - **Languages**: Kotlin for Android and Swift for iOS development, as they are officially supported by Google and Apple respectively.
    - **Frameworks**: For cross-platform development, consider frameworks like React Native or Flutter to build native-like mobile apps with a single codebase.
- **Queuing Systems:**
    - **Apache Kafka**: Kafka is a distributed streaming platform known for its high throughput, fault tolerance, and scalability. It can be used for real-time data processing and event-driven architectures.
    - **RabbitMQ**: RabbitMQ is a robust message broker that supports multiple messaging protocols (AMQP, MQTT, STOMP) and provides features like message queuing, routing, and delivery acknowledgments.

Databases

- **User Data:**
    - **Relational Database**: PostgreSQL or MySQL can be used to store user profiles, authentication credentials, and personal information. These databases offer ACID compliance, strong consistency, and support for complex queries.
- **Merchant Data:**
    - **NoSQL Database**: MongoDB or Couchbase or Cassandra can be used to store menu items, merchant details, and other semi-structured data. NoSQL databases provide flexibility in schema design, horizontal scalability, and high availability.
- **Order Management:**
    - **Relational Database**: PostgreSQL or MySQL can also be used for order management, storing information such as order status, items, prices, and foodie details. Relational databases offer transaction support, data integrity, and relational querying capabilities.
- **Full-Text Search:**
    - **Search Engine**: Elasticsearch or Apache Solr can be used for full-text search functionality, allowing foodies to search for merchants, menu items, or specific keywords. These search engines offer advanced search capabilities, relevance ranking, and support for complex queries.

APIs

Search APIs:

Search API to search for relevant merchants

Order APIs:

add-to-cart API

checkout API

payment API

track-order API

fetch-order API

order-status API

cancel-order API

Accounts/Profiles APIs:

create-profile API

update-profile API

update-address API

link-wallet API

Caching

Caching plays a vital role in improving performance, reducing latency, and minimizing load on back-end systems.

- **Content Delivery Networks (CDNs)**: Utilize CDNs to cache static assets such as images, CSS, and JavaScript files closer to users' geographical locations, reducing latency and offloading traffic from origin servers.
- **Result Caching**: Cache the results of computationally expensive operations or database queries to avoid redundant computations and improve response times for subsequent requests with identical inputs.
- **Query Result Caching**: Cache the results of database queries or API responses to serve identical queries or requests without hitting the back-end database or service, thereby reducing latency and database load.

Fault Tolerance

Fault tolerance is critical to ensure continuous operation and minimal disruption to users, even in the event of failures.

- **High Availability**: Ensure that the food delivery app remains available and responsive to users' requests, even in the presence of failures or disruptions.
- **Data Integrity**: Guarantee the integrity and consistency of data, such as user orders, payment transactions, and restaurant information, even during system failures or network partitions.
- **Resilience to Component Failures**: Design the system to withstand failures of individual components, such as servers, databases, or network infrastructure, without causing

complete service outage.
- **Graceful Degradation**: Provide a graceful degradation mechanism to ensure that the system can continue to operate with reduced functionality or performance during failures, rather than crashing completely.
- **Automatic Recovery**: Implement automated recovery mechanisms to detect and recover from failures quickly, minimizing manual intervention and downtime.

Loan Balancing

Load balancing is crucial to ensure optimal performance, scalability, and reliability. Load balancing should efficiently distribute requests to back-end servers based on factors like server health, response time, and available resources to ensure fast response times.

- **Scalability**: The load balancing solution should be able to scale horizontally to accommodate increasing traffic and user demand without affecting performance.
- **High Availability**: The system should ensure high availability by distributing incoming traffic across multiple servers to prevent overloading and minimize downtime.
- **Performance**: Load balancing should efficiently distribute requests to back-end servers based on factors like server health, response time, and available resources to ensure fast response times.
- **Fault Tolerance**: The load balancer should be resilient to failures, capable of detecting and rerouting traffic away from failed or unhealthy servers to maintain service availability.
- **Flexibility**: The system should support dynamic configuration adjustments to adapt to changing traffic patterns, server loads, and application requirements.

Security

All communication between different components needs to happen via HTTPS protocol to ensure security.

Authentication

Authentication Requirements

- **User Registration:** Allow actors to create accounts by providing necessary information like name, email, phone number, and password.
- **Password Management:** Implement secure password storage techniques like hashing with salt to protect user credentials.
- **Session Management:** Maintain user sessions securely to allow users to stay logged in across multiple interactions with the app.

Authentication Techniques

- **JSON Web Tokens (JWT)**: Use JWTs for stateless authentication. Upon successful login, issue a JWT containing user information and expiration time, which the client can include

in subsequent requests for authentication.
- **Secure Storage**: Store sensitive information such as passwords and tokens securely using encryption and hashing algorithms to prevent unauthorized access.
- **HTTPS**: Ensure that all communication between the client and server occurs over HTTPS to encrypt data in transit and prevent eavesdropping.

Authorization

Authorization Requirements

1. **Role-based Access Control (RBAC)**: Define roles such as foodie, pilot, and merchant, each with specific permissions to perform actions within the app.

Authorization Techniques

1. **Role-Based Access Control (RBAC)**: Assign predefined roles to users and grant permissions to each role based on their responsibilities. Use role hierarchy for more granular control.
2. **JWT Claims**: Include user roles and permissions as claims in JWTs, allowing the server to make access control decisions based on the user's claims.

Data Encryption

The sensitive data must be encrypted before saving it in the database for security purposes. Programming languages have different libraries for encrypting and decrypting sensitive data which is supposed to be serialized, stored or transmitted over network. For example, in Java we can generate a pair of public and private key using RSA algorithm. These keys should be stored in a secured vault and should be accessible for encrypting and decrypting the sensitive data. The public key is typically shared with external components and used for encryption. The private key remains confidential and is exclusively known to the service responsible for receiving and decrypting the data. This separation of keys ensures a robust and secure implementation in real-world applications. Implement TLS/SSL encryption for securing communication between clients and servers. TLS encrypts data in transit and provides authentication and data integrity mechanisms.

Conclusion

In conclusion, the design of the food delivery app outlined in this document aims to address the diverse needs of foodies, merchants and pilots while ensuring security, scalability, and reliability. By leveraging modern technologies, industry best practices, and robust architecture patterns, we have crafted a comprehensive solution poised to deliver exceptional user experiences and drive business growth.

The adoption of microservices architecture enables flexibility, scalability, and fault isolation, allowing the system to evolve and adapt to changing business requirements. The choice of

cloud-native technologies ensures high availability, elastic scalability, and cost efficiency, while containerization and orchestration provide consistency and manageability across environments.

Moreover, robust authentication and authorization mechanisms, along with data encryption techniques, safeguard sensitive information and ensure compliance with security standards and regulations. By prioritizing security and privacy, we uphold the trust of our users and protect the integrity of their data.