1. Hotel Booking System (like Expedia or Booking.com)

High-Level System Design

The hotel booking system should handle several core functions efficiently, including room availability, dynamic pricing, search and filtering, and payment processing.

Key Areas to Discuss:

1.1 Room Availability and Dynamic Pricing

- **Dynamic Pricing**: This involves adjusting prices based on factors like demand, season, location, and available rooms. A dynamic pricing model can use historical data and real-time demand to adjust pricing.
    - **Pricing Models**: Machine learning models can be implemented for price prediction.
    - **Availability**: The system must track room availability in real time.

Components:

- **Room Inventory**: Use a distributed system for managing room availability across multiple locations.
- **Dynamic Pricing Engine**: A microservice that adjusts prices based on predefined rules or machine learning algorithms.

1.2 Search and Filter Functionality

- **Search**: Users should be able to search for hotels based on various filters, such as location, price range, amenities, and ratings.
    - **Backend Search Engine**: Use **django orm** for fast, full-text search and filtering based on multiple parameters.

Components:

- **Search Service**: Microservice that performs searches, processes filters, and returns results.
- **Indexing**: Regularly update hotel data in the search index to reflect current availability and pricing.

1.3 Payment Processing and Reservation Confirmation

- **Payment Integration**: Integrating third-party services like **master card**, **mobile banking**, or **esewa/khalti gateway** for secure payment processing.
    - **Transactional Integrity**: Ensure that payments and reservations are processed atomically to avoid double booking.

Components:

- **Payment Gateway Service**: Handles secure payment requests, validation, and processing.
- **Reservation Service**: Once payment is processed, this service updates room availability and confirms the reservation.

1.4 Managing Bookings, Cancellations, and Modifications

- **Booking Management**: Users should be able to view, modify, or cancel their bookings.
    - **Cancellation Fees**: Implement rules for cancellations and modify bookings (e.g., charge users a cancellation fee depending on the time of cancellation).

Components:

- **Booking Service**: Manages reservations, cancellations, and modifications.
- **Email/SMS Notifications**: Sends confirmations, reminders, and cancellation notices.

1.5 Ratings and Reviews for Hotels and Services

- **User Reviews**: Allow users to submit ratings and reviews for hotels and services.
    - **Moderation**: Implement moderation to ensure quality of reviews.

Components:

- **Review Service**: Handles the submission and retrieval of ratings and reviews.
- **Database**: Store reviews in a database depending on the scale of the platform.

1.6 Scalability Considerations

- **Horizontal Scaling**: Break down the system into microservices for booking, search, payments, etc., and scale each independently.
- **Caching**: Use caching solutions like **Redis** to speed up frequent operations like room availability and search results.
- **Load Balancing**: Use **NGINX** to distribute traffic evenly across multiple service instances.
- **Multilingual Support**: Use internationalization (i18n) practices to support multiple languages and currencies.

Example Technologies:

- **Frontend**: React for web apps, Flutter for mobile apps.
- **Backend**: python(Django/fastapi/flask).
- **Database**: PostgreSQL for transactional data, Elasticsearch for search, Redis for caching.
- **Payment Gateway**: master card, mobile banking, or esewa/khalti gateway.

---

2. URL Shortener System (like Bit.ly)

High-Level System Design

A URL shortener is a system that converts long URLs into shorter, more manageable links. It also handles redirection when these short URLs are visited.

Key Areas to Discuss:

2.1 Shorten URLs

- **Generate Unique Short URL**: The system should generate a unique, short URL that maps to the original URL.
    - **Unique ID Generation**: The short URL can be created using algorithms like **Base62 encoding**, **hashing (SHA256)**, or **random strings**.

Components:

- **Shortening Algorithm**: Generate short, unique URLs for each long URL input.
- **Database**: Store mappings of short URLs to long URLs in a high-performance database like **Redis** or a key-value store.

2.2 Redirect URLs

- **Redirection**: When a user accesses a short URL, the system should redirect them to the original URL.
    - **Database Lookup**: The system should query a fast database to find the long URL corresponding to the short URL.

Components:

- **Redirection Service**: Handles redirect requests from users accessing short URLs.
- **Cache**: Use **Redis** to cache frequently accessed short URLs to improve performance.

2.3 Scalability Considerations

- **High Availability**: The URL shortener must handle billions of short URLs and provide low-latency redirection.
    - **Database**: Use **NoSQL databases** like **DynamoDB**, **Cassandra**, or **Redis** for fast lookups.
    - **Load Balancing**: Use load balancers to distribute traffic across multiple service instances.
    - **Rate Limiting**: Implement rate limiting to prevent abuse (e.g., spam or DDoS attacks).

2.4 Collision Handling

- **Collisions**: When generating short URLs, ensure that the generated short URL does not conflict with an existing one.
    - **ID Collision Handling**: One way to handle collisions is to use a combination of **hashing** and **Base62 encoding**.
    - **Unique ID Space**: Ensure that the generated short URL is unique by checking the database for collisions before finalizing the short URL.

2.5 Optional Features

- **URL Expiration**: Allow short URLs to expire after a certain period.
    - **Metadata**: Track analytics like the number of clicks, referring domains, etc.
    - **Custom Short URLs**: Allow users to customize the short URL instead of using random IDs.

Components:

- **Analytics Service**: Collects data about each short URL, such as number of clicks, geographic location of users, and referral sources.
- **Expiration Mechanism**: Use time-based expiration logic to delete old short URLs from the database.

2.6 Scalability Considerations

- **High Traffic**: The system must handle high volumes of URL shortening and redirection requests.
    - **Distributed Systems**: Use a distributed key-value store like **Redis**, **DynamoDB**, or **Cassandra** for fast lookups and scalability.
    - **Horizontal Scaling**: Use **Microservices** for URL shortening, redirection, analytics, etc., and scale each service as needed.
    - **Caching**: Frequently accessed short URLs should be cached to reduce database load and improve performance.

Example Technologies:

- **Frontend**: React for the web interface.
- **Backend**: Python (Flask/Django).
- **Database**: Redis for caching and high-speed lookups, postgres for long-term storage.
- **Analytics**: Store click data in **Elasticsearch** or **MongoDB**.