Design Flight Booking

Functional
1. Users should be able to search for flight with source airport, destination airport, select one way vs round trip, start and return date, class of flight (example Economy, Economy plus, Business etc).

2. User should be able to book or cancel flights. Users should be able to select seat and hold the reservation untill the payment is completed. There will be a 15 minute payment time window. If the user is unable to pay within that time window, release the seat.

3. Prevent double booking

4. System should send flight notifications and reminders and generate eBoarding pass for users.

5. Optional (ML for Price Flight drop prediction)

Non Functional
1. Strong consistancy on flights booking.
2. Flight search latency should be less than 200ms

Capacity Estimation
1. 4 billion daily trevallers(half of the world polulation, around 10,000 cities are there in the world 40, 000 travellers per city

10,000×400,000 travelers/day×10 KB=40,000,000,000 KB/day=40 TB/day

Core Entities / Services

1. **User**: (user_id, name, email, phone, preferences)
2. **Flight**: (flight_id, airline, source, destination, departure_time, arrival_time, price, class)
3. **Booking**: (booking_id, user_id, flight_id, status, payment_info, seat_number, hold_expiry_time)
4. **Payment**: (payment_id, booking_id, amount, status)
5. **Notification**: (notification_id, user_id, type, content, timestamp)

User Service, Flights Service, Booking Service, Payment Service, Notification Service

API Design

Flight Search

GET /api/flights?source={source}&destination={destination}&date={date}&class={class}&round_trip={true/false}

Booking

POST /api/bookings

Payload:

{

"user_id": "123",

"flight_id": "987",

"class": "Economy",

"seat_number": "12A",

"payment_info": { ... }

}

Cancel Booking

DELETE /api/bookings/{booking_id}

Get Booking

GET /api/bookings/{booking_id}

Get Eboarding Pass

GET /api/boarding-passes/{booking_id}

High Level Overiew

The user searches for flights, and the system fetches available flights from the database. To ensure scalability, the **Flights Database** is **sharded** and uses **read replicas** for faster queries. Once the user selects a flight, the seat is locked in Redis for 15 minutes to prevent double booking. The booking details are temporarily stored in the **Booking Database**, which is also **sharded** for scalability, but the seat isn't confirmed yet. The system then redirects the user to a third-party payment service. After the user completes the payment, the **Payment Service** receives a callback from the third-party gateway indicating whether the payment was successful or failed. If successful, the **Booking Database** is updated with the confirmed seat, and the lock in Redis is released. If the payment fails or the 15-minute window expires, the seat lock is released, and the seat becomes available again. Notifications about booking status are sent via Kafka to inform the user about the success or failure of the booking.

In this architecture, the **Flights Database** requires read replicas for scalability, while the **Booking Database** and **Payment Database** can be sharded to handle high volumes of data.