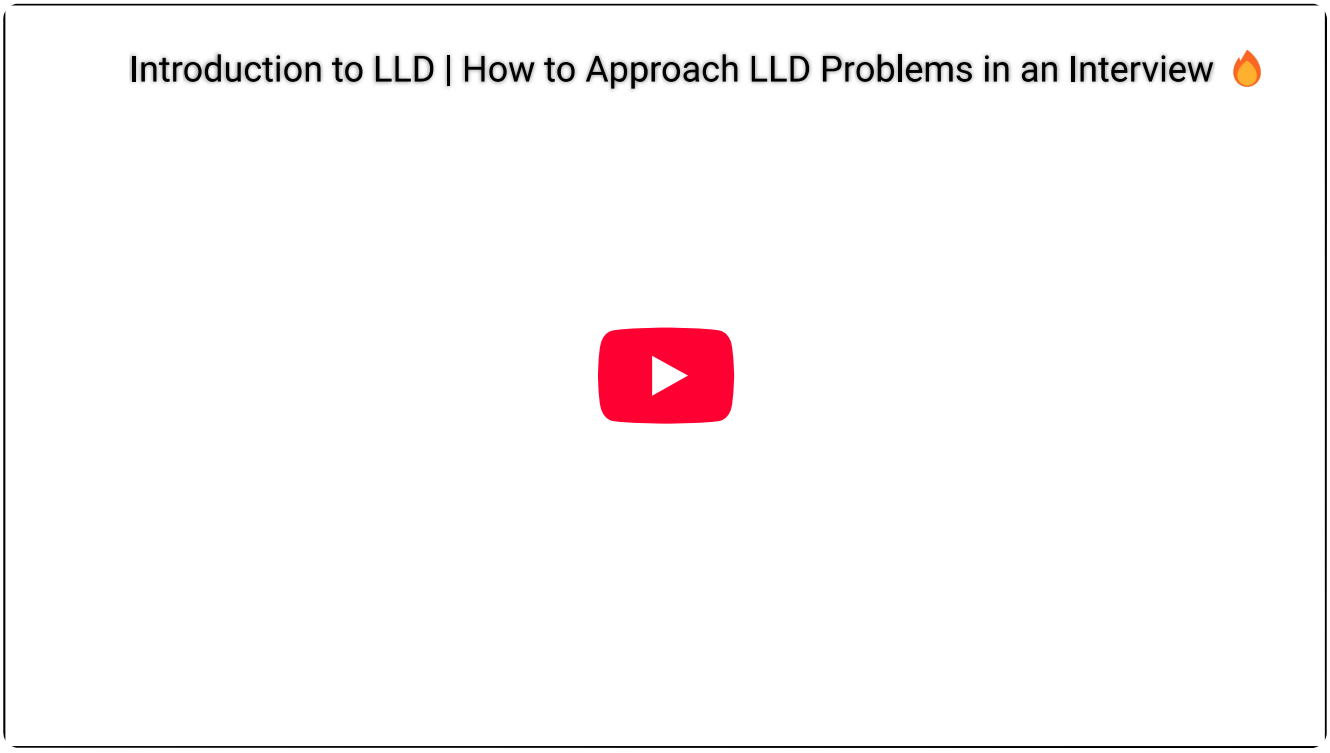
How to Approach LLD Problems



Topic Tags:

System Design LLD

🎯 Low-Level Design (LLD) Interview Approach

Low-Level Design (LLD) problems often form a critical component of technical interviews, especially for backend engineering roles. They evaluate your ability to design software systems at a detailed level, focusing on:
1. Class structures 🏗️,
2. Design patterns 🧩,
3. Relationships 🔗,
4. Implementation details 🛠️.
Here's a structured approach to tackle LLD problems effectively, using a movie ticket booking system as an example: 🎥🎟️

1) Clarify Requirements and Use Cases 📋

The interview begins with a brief introduction, where the interviewer explains the format and provides a high-level overview of the system you'll design. Afterward, the problem statement is presented. Here's how the conversation might unfold:

👨‍💼 **Interviewer:** Design a Movie ticket booking system

👨‍💻 **Interviewee:** Absolutely! Based on the problem, here's my understanding of the system so far:
• Users will browse theaters (then screens), select movies or shows, book seats, and make payments. 🎬💳
• The system needs to handle concurrent bookings efficiently. 🔄
Would you like me to add any details to this understanding?

👨‍💼 **Interviewer:** That's a good summary. We can move ahead.

👨‍💻 **Interviewee:** To clarify further:
• Should we support multiple user types, like admins and customers? 👤👨‍💼
• How do we handle seat availability during concurrent bookings? ⏳
• Is payment processing and ticket generation included in scope? 💰🎟️

👨‍💼 **Interviewer:** Yes, concurrency and payment are key. Admin roles and advanced features are out of scope for now.

👨‍💻 **Interviewee:** Understood. Here's the core flow I'll focus on:
1. Users search for movies based on location 📍
2. Movies are mapped to screens and screens to theaters 🎥🏛️
3. Seats are selected and locked for booking 🔒🪑
4. Payment processing confirms the booking 💳

Does this cover the main use cases?
Yes, that looks great. Proceed ahead.

> *NOTE: Always write down key points discussed with the interviewer. This helps maintain clarity and ensures you address the critical aspects of the problem.* 📝

2) Identify Entities (Classes) 🏷️

Once the requirements and the use-cases are clear, break the system down into core entities.

👨‍💻 **Interviewee:** Based on the requirements, I've identified these core entities:
• User 👤
• Movie 🎬
• Theater 🏛️
• Show 📅
• Seat 🪑

• Booking 🎟️
• Payment 💳

👨‍💼 **Interviewer:** Yes, that looks great. Proceed ahead.

> *NOTE: Avoid excessive entity details as they consume time better spent explaining the core problem's solution.* ⏱️

3.) Define Relationships Between Entities 🔄

Once the entities are clear, proceed with mapping their relationships and interdependencies.

👩‍💼 **Interviewee:** Here's how the entities are related:
1. A Theater has multiple Screens, each linked to specific Seats. 🏛️ ➡️ 🖥️ ➡️ 🪑
2. A Show occurs on a specific Screen and maps to a Movie. 🎬 🎥
3. A Seat is tied to a Show for tracking availability during bookings. 🪑 📅
4. A Booking associates a User with Seats, Show, and Payment. 🎟️

For example:
• A Show has a List to track seat availability during that specific screening.
• A Show is also uniquely associated with a specific Movie and Screen.
Below is the example implementation of a Show class:

Java

```java
public class Show {
    private final String id;
    private final Movie movie;
    private final Screen screen;
    private final Date startTime;
    private final Integer durationInMinutes;
}
```

• A Theater contains multiple Screens.
• Each Screen is responsible for managing its Seats.

Java

```java
public class Theater {
    private final String id;
```

```
    private final String name;
    private final String location;
    private final List<Screen> screens;
}
public class Screen {
    private final String id;
    private final String name;
    private final List<Seat> seats;
}
```

• A Booking includes details of the User, Show, and Seats booked.
• Each Booking is linked to a Payment to track the transaction status.

Java

```
public class Booking {
    private final String id;
    private final User user;
    private final Show show;
    private final List<Seat> bookedSeats;
    private final Payment payment;
}
public class Payment {
    private final String id;
    private final double amount;
    private final PaymentStatus status;
    private final Date timestamp;
}
```

👨‍💼 **Interviewer:** Yes, this is aligned. Please proceed ahead.

4) Identify Core Methods for Each Class 🔑

Once the relationships are identified, the next step is to focus on the critical methods that drive the system's functionality and where they should be put to support a clean code architecture. These methods handle the primary actions like booking seats, managing availability, and ensuring concurrency in a multi-user environment.

👨‍💼 **Interviewee:** Moving on to the primary functionalities, I'll focus on the major methods which are critical to the system:

• Seat Management 🪑
○ lockSeats(List<Seat> seats, int showId) – Locks selected seats for a user, preventing others from booking them during the session. 🔒
○ releaseSeats(List<Seat> seats, int showId) – Releases previously locked seats if the booking is canceled or not confirmed. 🔓

• Booking Management 🎟️
○ createBooking(int userId, int showId, List<Integer> seatIds) – Creates a new booking for the user with the selected seats. 📅
○ cancelBooking(int bookingId) – Cancels a booking and releases the locked seats. ❌

• Concurrency Handling ⏳
○ To manage concurrency, we can use synchronized blocks or distributed locks to ensure that multiple users cannot lock the same seats at the same time. Additionally, implementing a retry mechanism (e.g., exponential backoff) in case of seat locking failures can further improve the robustness of the system. 🔒

• Design Patterns 🧩
○ Strategy Pattern for handling different payment methods. This allows the system to switch between different payment methods (credit card, wallet, etc.) without modifying the booking logic. 💳
○ Factory Pattern to create different types of bookings (VIP, Regular, etc.) based on user preferences or seat categories. The Factory pattern will centralize object creation and make the system easier to scale with minimal changes. 🏭

👨‍💼 **Interviewee:** Would you like me to dive deeper into any particular functionality?

👨‍💼 **Interviewer:** Let's focus on the lockSeats and createBooking methods.

> *Note: Focus on implementing core business logic rather than basic CRUD operations. The interviewer primarily evaluates your approach to designing and implementing complex system functionalities.*

5) Implement Necessary Methods 💻

👨‍💼 **Interviewee:** I'll begin with the lockSeats method. Would you prefer I explain the approach first or code while explaining?

👨‍💼 **Interviewer:** You may code and explain simultaneously.

👨‍💼 **Interviewee:** I'll proceed with the implementation.

> *While coding, maintain a clear structure and use design principles like SOLID, DRY, KISS, YAGNI, etc. Explain your choices to the interviewer like what all design patterns you can use and why they are used, validate your approach, and ensure alignment with the overall architecture.*

Java

```java
public boolean lockSeats(List<Integer> seatIds, int showId, int userId) {
  synchronized (this) {
    for (int seatId : seatIds) {
      Seat seat = seatRepository.getSeat(seatId, showId);
      if (seat.getStatus() != SeatStatus.AVAILABLE) {
        return false; // Seat already locked or booked
      }
      seat.setStatus(SeatStatus.LOCKED);
      seat.setLockedBy(userId); // Associate the locked seat with the user
      seatRepository.updateSeat(seat);
    }
  }
  return true;
}
```

> *Once the interviewer approves your approach, proceed with implementing the next method.*

👨‍💼 **Interviewer:** We can proceed with the createBooking() method implementation now.

👨‍💻 **Interviewee:** Yes sure.

Java

```java
public Booking createBooking(final String userId, final Show show, final L
    // Check if any seat is already booked
    if (isAnySeatAlreadyBooked(show, seats)) {
        throw new SeatPermanentlyUnavailableException();
    }
    // Attempt to lock the seats for the user
    boolean lockSuccess = seatLockProvider.lockSeats(seats, show.getId(),
```

```
        if (!lockSuccess) {
            throw new SeatLockFailedException("Failed to lock seats for the us
        }
        // Generate a unique booking ID
        final String bookingId = UUID.randomUUID().toString();
        final Booking newBooking = new Booking(bookingId, show, userId, seats)
        // Store the booking
        showBookings.put(bookingId, newBooking);
        // Return the new booking
        return newBooking;
    }
```

6) Exception Handling ⚠️

Incorporate error handling to manage edge cases gracefully. Proper error handling ensures the system behaves predictably even under unexpected conditions, enhancing the user experience and system reliability.

Examples:

○ No Seats Available: Ensure that users are informed when attempting to book seats that are already reserved. ❌🪑

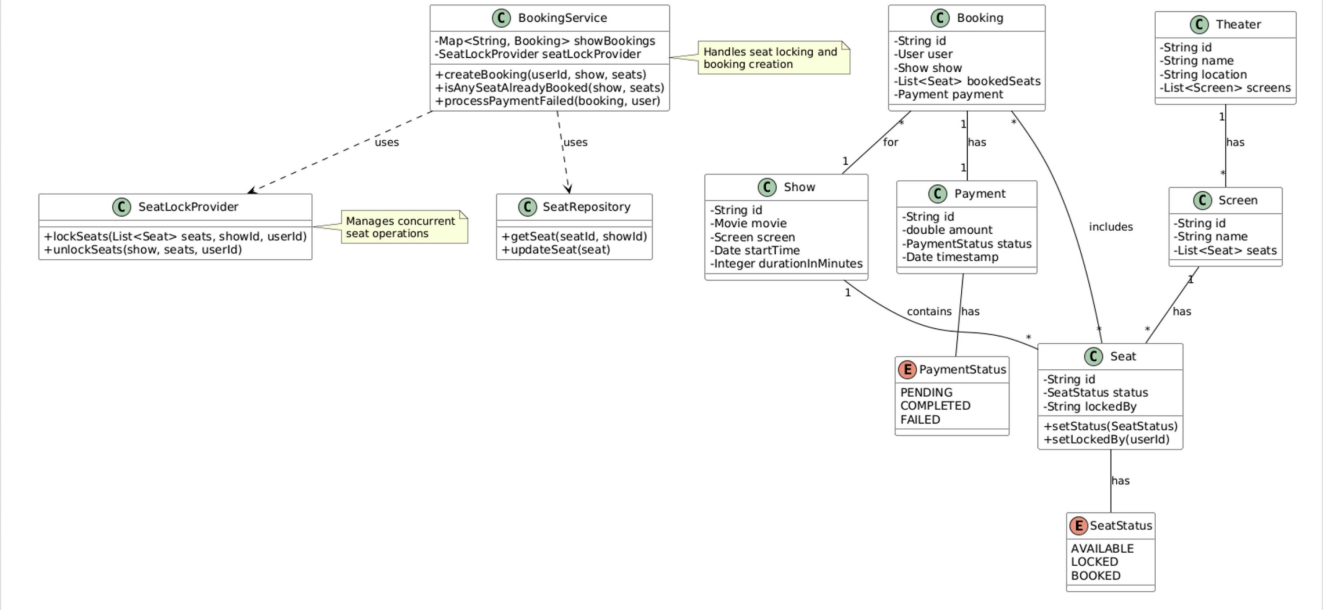
Java

```
private boolean isAnySeatAlreadyBooked(final Show show, final List<Seat> sea
  final List<Seat> bookedSeats = getBookedSeats(show);
  for (Seat seat : seats) {
    if (bookedSeats.contains(seat)) {
      return true;
    }
  }
  return false;
}
public Booking createBooking(final String userId, final Show show, final Lis
  if (isAnySeatAlreadyBooked(show, seats))
    throw new SeatPermanentlyUnavailableException();
  // rest of the logic
}
```

Payment Failure: Handle failed transactions and provide options for retrying or canceling the booking process. 💳🚫

Java

```java
public void processPaymentFailed(final Booking booking, final String user)
  if (!booking.getUser().equals(user)) {
    throw new BadRequestException();
  }
  if (!bookingFailures.containsKey(booking)) {
    bookingFailures.put(booking, 0);
  }
  final Integer currentFailuresCount = bookingFailures.get(booking);
  final Integer newFailuresCount = currentFailuresCount + 1;
  bookingFailures.put(booking, newFailuresCount);
  if (newFailuresCount > allowedRetries) {
    seatLockProvider.unlockSeats(booking.getShow(), booking.getSeatsBooked
  }
}
```



🏁 Conclusion:

The art of Low-Level Design interviews lies not just in technical knowledge, but in demonstrating a methodical approach to problem-solving. Success comes from:

1. Systematic Progression

- Begin with thorough requirement gathering
- Move systematically from high-level entities to detailed implementations
- Validate your approach at each step with the interviewer ✅

2. Practical Considerations

- Focus on maintainable and scalable solutions
- Address critical aspects like concurrency and error handling
- Demonstrate knowledge of design patterns where relevant 🧠

3. Communication Excellence

- Clearly articulate your design choices
- Explain tradeoffs in your decisions 💭
- Show adaptability when receiving interviewer feedback 🌱

Remember:

A successful LLD interview is not about creating a perfect design in the first attempt. Instead, it's about demonstrating:

- Your structured thinking process
- Your ability to translate requirements into concrete solutions
- Your understanding of object-oriented design principles 💡
- Your skills in writing clean, maintainable code 🖌️
- Your openness to feedback and ability to iterate on your design 🔄

By following this systematic approach and maintaining clear communication throughout the interview, you'll be well-equipped to tackle any Low-Level Design challenge effectively. 💪🎯