perplexity

1. Interview-Style Opening

"This is a critical scenario in Fintech. Double spending destroys trust and loses money, so my approach relies on **layers of defense**. I don't trust the application code alone; I enforce correctness at the database level and ensure safety at the API level.

I generally use a strategy of **Optimistic Locking** for performance combined with **Idempotency Keys** for safety."

2. Problem Understanding and Clarification

The goal is to ensure that if two threads (or two different microservice instances) try to deduct money from the same wallet at the same time, only one succeeds.

Scenario:

- Wallet Balance: $100
- Thread A: Deduct $50
- Thread B: Deduct $50
- Result must be $0, not $50 (Race Condition).
- Result must not be -$50 (Double Spending).

3. High-Level Approach (Concurrency Control)

1. **Database Level (The Source of Truth):**
    - **Pessimistic Locking (`SELECT ... FOR UPDATE`):** Locks the row. Safe but slow. Good for high contention (e.g., Apple's bank account).
    - **Optimistic Locking (`version` column):** No database locks. Fails if the data changed since we read it. Much faster for typical user wallets.
2. **API Level (Idempotency):**
    - Clients must send a unique `Idempotency-Key` (UUID) with every payment request.
    - We store this key. If the client retries (network timeout), we check the key. If it exists, we return the *previous* success response instead of charging again.

4. Visual Explanation (Optimistic Locking)

Diagram: The Version Check

```
DB State: [ User: Alice, Balance: 100, Version: 1 ]

Thread A (Read):             Thread B (Read):
Reads Bal=100, Ver=1         Reads Bal=100, Ver=1

Thread A (Write):
UPDATE wallet
SET balance = 50, version = 2
WHERE id = Alice AND version = 1;
Result: 1 row updated (Success)

Thread B (Write):
UPDATE wallet
SET balance = 50, version = 2
WHERE id = Alice AND version = 1;
Result: 0 rows updated (FAIL!) -> Because version is now 2 in DB
```

Explanation: Thread B fails because the `WHERE` clause condition (`version = 1`) is no longer true. We catch this failure and either retry or throw an error.

5. Java Code (Optimistic Locking with JPA)

In Spring Data JPA, this is trivially implemented using the `@Version` annotation.

```
@Entity
public class Wallet {
    @Id
    private Long id;

    private BigDecimal balance;

    @Version
    private Long version; // The magic field
}

@Service
public class PaymentService {

    @Autowired
    private WalletRepository walletRepo;

    @Transactional
    public void processPayment(Long walletId, BigDecimal amount) {
        Wallet wallet = walletRepo.findById(walletId)
                .orElseThrow(() -> new RuntimeException("Wallet not found"));

        if (wallet.getBalance().compareTo(amount) < 0) {
            throw new InsufficientFundsException();
        }
```

```
        // JPA automatically checks the version on save
        // Executes: UPDATE wallet SET balance=?, version=2 WHERE id=? AND version=1
        wallet.setBalance(wallet.getBalance().subtract(amount));

        try {
            walletRepo.save(wallet);
        } catch (ObjectOptimisticLockingFailureException e) {
            // Handle the race condition
            throw new ConcurrentTransactionException("Please retry transaction");
        }
    }
}
```

6. The "Double Spend" via Retry (Idempotency)

Optimistic locking handles *simultaneous* threads. But what if the user clicks "Pay" twice? Or the first request times out but actually succeeded?

We need an **Idempotency Filter**.

```
@RestController
public class PaymentController {

    @PostMapping("/pay")
    public ResponseEntity<?> pay(@RequestHeader("Idempotency-Key") String key,
                                 @RequestBody PaymentRequest req) {

        // 1. Check if key exists in Redis/DB
        if (transactionRepo.existsByIdempotencyKey(key)) {
            return ResponseEntity.ok(transactionRepo.findByKey(key).getResult());
        }

        // 2. Process (Optimistic Locking happens here)
        PaymentResult result = paymentService.processPayment(req);

        // 3. Save Key + Result
        transactionRepo.save(new IdempotencyRecord(key, result));

        return ResponseEntity.ok(result);
    }
}
```

7. How to Explain This to the Interviewer

"To prevent double spending, I use a two-pronged approach.

First, I implement **Optimistic Locking** using a version column in the database. This prevents two concurrent threads from overwriting each other's updates. If a collision occurs, the database rejects the second write, and I can catch that exception to trigger a retry or inform the user.

Second, to handle network retries or 'double clicks', I enforce **Idempotency**. I require the client to send a unique key. I check this key before processing. If we've already seen it, I return the cached result. This ensures that even if I receive the same request 10 times, I only deduct the money once."

8. Edge Cases (Pessimistic Locking)

Q: When would you NOT use Optimistic Locking?
*A: If I'm designing the system for a 'Hot Wallet' (e.g., the merchant account receiving 1000 payments/sec). Optimistic locking would cause 999 retries for every 1 success. In that specific case, I would use **Pessimistic Locking** (`SELECT ... FOR UPDATE`) to queue the requests at the database level, or better yet, use an append-only ledger architecture.*

⁂

1. https://codexplo.wordpress.com/2015/12/07/longdouble-are-not-atomic-in-java/
2. https://www.computer.org/publications/tech-news/trends/idempotency-in-payment-processing-architecture
3. https://www.khilan.com/ic/unpublished/outsourcingreport.pdf
4. https://stackoverflow.com/questions/129329/optimistic-vs-pessimistic-locking
5. https://www.geeksforgeeks.org/system-design/airbnb-idempotency-avoiding-double-payments-in-a-distributed-payments-system/
6. https://codefinity.com/courses/v2/64fdb450-1405-4e74-8cd4-45fc2ebd37e5/3ff06e21-8666-4d2f-970c-0768ce6e429f/363780af-2b0e-4760-853c-c049b14ac166
7. https://www.moderntreasury.com/learn/pessimistic-locking-vs-optimistic-locking
8. http://jeremymanson.blogspot.com/2007/08/atomicity-visibility-and-ordering.html
9. https://stackoverflow.com/questions/29280857/java-synchronisation-atomically-moving-money-across-account-pairs
10. https://dzone.com/articles/longdouble-are-not-atomic-in-java
11. https://people.csail.mit.edu/shanir/publications/MultiProg-journal.pdf
12. https://vivekbansal.substack.com/p/optimistic-vs-pessimistic
13. https://dev.to/budiwidhiyanto/ensuring-reliable-payment-systems-with-idempotency-2d0l
14. https://travisdowns.github.io/blog/2020/07/06/concurrency-costs.html
15. https://inery.io/blog/article/optimistic-vs-pessimistic-locking-difference-and-best-use-cases/