Laravel Interview Task — Flash-Sale Checkout (Concurrency & Correctness)

Summary

Build a **small API** that sells a limited-stock product during a flash sale. It must handle **high concurrency** without overselling, support **short-lived holds**, **checkout**, and an **idempotent payment webhook**. No UI.

Target: **Laravel 12**, **MySQL (InnoDB)**, **any Laravel cache driver** (database/file/memcached/redis/etc.).

Core Requirements

1. Product Endpoint

- Seed one product with price and finite stock.
- GET /api/products/{id} returns basic fields and **accurate available stock**.
- Must stay fast under burst traffic **and** remain correct when stock changes.

2. Create Hold

- POST /api/holds { product_id, qty } creates a **temporary reservation** (~2 minutes).
- Success → { hold_id, expires_at }.
- Holds **immediately reduce availability** for others.
- Expired holds must **auto-release** availability (not only on read).

3. Create Order

- POST /api/orders { hold_id } creates an order in a pre-payment state.
- Only valid, unexpired holds; each hold can be used once.

4. Payment Webhook (Idempotent & Out-of-Order Safe)

- POST /api/payments/webhook with an **idempotency key** updates order to paid on success, or cancels and releases on failure.
- Webhook may arrive **multiple times** and **before** the client receives the order response. Final state must be correct.

What’s a “webhook” in this task? (for juniors)

A **webhook** is an HTTP request that **another system sends to your API** to notify that an event occurred.

In this task, imagine a payment provider finishing a card payment. Instead of the app polling “is it paid yet?”, the provider **calls your endpoint** (e.g., POST /api/payments/webhook) with the payment result. The goal is to **trust but verify**, update the order’s status, and ensure repeating the same webhook doesn’t break anything.

Why it’s tricky

- **Duplicates:** The provider might send the **same webhook multiple times** (retries/timeouts).
- **Out of order:** The webhook might arrive **before** the API finishes creating the order response.
- **At-least-once delivery:** The system must handle receiving it **once or many times** and still end up with the **same correct final state**.

Idempotency (tiny hint)

Idempotency means: handling the same request more than once should have the same effect as handling it once.
In practice: use a unique **idempotency key** to **recognize duplicates** and make the handler **safe to run multiple times** without double-changing order state or stock.

Non-Functional Expectations

- **No overselling** under heavy parallel requests.
- Handles **deadlocks** and **race conditions** robustly (prove via tests/metrics).
- **Caching** improves read performance **without stale/incorrect stock** (use any Laravel cache driver, including the **database cache**).
- Background processing for **expiry** that won’t double-run or miss items.
- Avoid **N+1** in any list endpoint.
- **Structured logging/metrics** around contention, retries, and webhook dedupe.

Constraints

- **API only.**
- Use **MySQL** and **any Laravel cache driver**.
- Don’t use heavy “solve-it-for-me” libraries for the core problems (concurrency/idempotency).
- Keep the codebase compact and readable.

Deliverables

1. **Repository** with migrations, seeders, and minimal code to run locally.
2. **README (≤2 pages)** covering:
   - Assumptions and invariants enforced.
   - How to run the app and tests.

    - Where to see logs/metrics.
3. **Automated Tests** showing:
    - Parallel hold attempts at a stock boundary (no oversell).
    - Hold expiry returns availability.
    - Webhook **idempotency** (same key repeated).
    - Webhook arriving **before** order creation.