# Technical Assessment: The High-Concurrency Inventory System

**Time Limit:** 2-3 Hours

## Context

We are launching a flash-sale platform. A common problem in our industry is "overselling"—selling more inventory than we actually have due to race conditions when thousands of users click "Buy" simultaneously.

## The Challenge

Implement a robust, thread-safe inventory system in Python that guarantees **Strict Consistency**.

## Requirements

### 1. The Scenario

- You have an inventory table with **Item A**.
- Total Stock: **100 units**.
- Traffic: **1,000 concurrent requests** attempting to buy Item A.

### 2. The API

Create an endpoint POST /buy_ticket that:

1. Checks if inventory is > 0.
2. Decrements inventory by 1.
3. Records a "Purchase Record" in a separate table.
4. Returns 200 OK if successful, 410 GONE if sold out.

### 3. Critical Constraints

- **Zero Overselling:** Under no circumstances can the final inventory count be negative.
- **Zero Underselling:** If there is stock, a valid request must succeed (deadlocks should be handled or avoided).
- **Concurrency:** Your solution must work correctly even if the application is deployed across multiple worker processes (e.g., Gunicorn with 4 workers). *Hint: In-memory Python locks (threading.Lock) will not work across multiple processes.*

## Deliverables

1. **app.py:** The application logic.

2. **proof_of_correctness.py:** A script that spawns 50+ concurrent threads/processes to hit your endpoint and verifies that exactly 100 items were sold, not 101 or 99.

## Grading Criteria

- **Database Locking:** Proper use of transactions, row-level locking (SELECT FOR UPDATE), or atomic increments.
- **Error Handling:** How do you handle database contention/timeouts?