Design Inventory system

场景：
你需要设计一个库存管理系统（Inventory System），用于管理商品库存和订单操作，保证库存在高并发环境下的正确性，同时考虑系统可扩展性和安全性。

核心功能要求

1. **Add stock（增加库存）**
    - **支持补货或增加仓库库存数量。**
2. **Reserve stock（预留库存）**
    - **当用户下单时，将库存临时锁定，防止其他订单抢占（防止超卖）。**
3. **Fulfill stock（完成订单）**
    - **当订单完成支付或发货时，真正扣减库存。**
4. **Clear reservation（清理预留）**
    - **订单取消或超时未支付时，将预留库存释放回可用库存。**

关键非功能点

1. **并发控制 / race condition**
    - **多个用户同时下单时，需要防止库存超卖。**
    - **面试官可能希望你讨论事务、锁、乐观/悲观并发控制或 Redis 原子操作。**
2. **扩展性 / Scaling**
    - **系统需要支持高并发访问。**
    - **可考虑缓存、分库分表、异步消息队列等。**
3. **安全性**
    - **防止非法库存操作。**
    - **API 鉴权、权限控制、加密传输等。**
4. **Production Deployment Best Practices**
    - **高可用 DB、负载均衡、日志监控、备份、CI/CD 部署。**

面试考察重点

- **业务流程理解：Add → Reserve → Fulfill / Clear**

- 事务与并发处理：防止库存被重复预留或扣减
- 系统可扩展性：缓存、异步处理、分库分表
- 安全性与运维：权限控制、监控、高可用部署

inventory management 是 这个 instacart blog 里提到的 https://tech.instacart.com/how-instacarts-item-availability-evolved-over-the-pandemic-5ce8e84468a0

系统要准确地反映每个 product 的 real-time availability，保证 customer 下单的话 就能拿到他们选购的商品。

两个 role：customer 下单；shopper 接单-》去对应商店取货-》送货

Design Category system

场景

- 设计一个 商品分类展示系统（Catalog / Category System），用户通过网页浏览商品。
- 商品可以属于 多个分类（多对多关系）。
- 分类可以有 多级层次（父分类、子分类），但前端每次只需要展示 直属子类。

用户需求

1. 用户打开网页时：
    - 显示所有顶层分类（例如 Food、Clothes）。
    - 显示部分商品（热门商品或顶层分类商品）。
2. 用户点击某个分类：
    - 显示该分类的 直属子分类。
    - 显示该分类下的 商品。
    - 不需要递归显示所有子孙分类（简化查询和缓存）。
3. 商品可能属于 多个分类：
    - 一个商品可能在不同分类页面同时出现。

面试考察点

1. 数据库设计

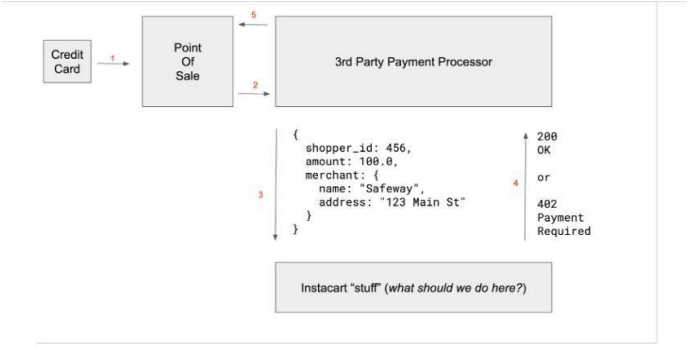
- 如何存储分类（Category 表自关联实现树形结构）。
- 如何存储商品与分类的多对多关系（Product / ProductCategory 表）。

2. API 设计
   - 如何设计请求与返回格式。
   - 如何让前端方便地 fetch 分类和商品信息。
3. 数据访问模式
   - 如何查询顶层分类。
   - 如何查询某分类下的直属子分类和商品。
   - 如何处理商品属于多个分类的情况。
4. 系统扩展性 / 高可用性（follow-up）
   - 如何缓存分类和商品信息。
   - 如何在大量用户访问时扩展系统。
   - 如何保证系统稳定运行。

重点注意

- 前半部分面试会偏重 API + 数据库 schema。
- 后半部分可能会问 整体系统架构、Cache、Load Balancer、数据同步策略 等细节。
- 面试中需要清楚表达：
  1. 数据在数据库中的存储形式。
  2. 请求格式和响应格式。
  3. 前端 fetch 的逻辑。

Design Payment Service



At Instacart, we have customers that place orders on our website, and we hire personal shoppers whose job it is to go to grocery stores to fulfill those orders.

We give credit cards to our shoppers so they can purchase groceries for orders we've assigned them. Our servers receive an HTTP request from our payment processor every time a shopper swipes the credit card that we give them. The payload looks like this:

```
{
  shopper_id: 456,
  amount: 100.0,
  merchant: {
    name: "Safeway",
    address: "123 Main St"
  }
}
```

When we receive the HTTP request, we have to synchronously respond within 1 second with a 200 OK to approve the transaction or 402 Payment Required to decline the transaction.

Say you are hired tomorrow, and you're leading the three person team in this room. How would you suggest we build the application that processes these requests? Some areas we should be sure to cover are **physical infrastructure, data stores, data model, security, and performance considerations**. For simplicity, we should start off with the assumption that 1 Shopper has just 1 Order at 1 merchant.

题目理解 & 重点

- **场景**：Instacart 给 shopper 一张信用卡。每次刷卡，**第三方支付处理器**会向我们的 **Payment Verification API** 发一个 HTTP 请求（shopper_id, amount, merchant{ name,address } ...）。
- **SLA**：**必须在 1s 内**同步返回：200 OK（批准）或 402 Payment Required（拒绝）。
- **V1 简化假设**：1 个 shopper 同时只服务 1 个 order，且只在 1 家 merchant 购物。
- **考察重点**：API 设计、数据模型、数据存储、**安全**、**性能**、**幂等**、**一致性/事务**、**扩展性（DB 与服务）**、**稳定性/容错**、以及**误差/风险判断**。

一句话总体方案

用 **无状态的 Payment Verification Service** 承接 PSP（支付处理器）回调；**L1 本地缓存 + L2 Redis（写穿/写通）** 保存“**每个 shopper 的当前订单核验快照**”；**PostgreSQL** 做强一致落库

与审计；**决策引擎**在 1 次 Redis 读 + 1 次短事务或原子脚本内完成额度/商户核验并生成**幂等**决策；**异步事件队列**做风控学习与对账。全链路 mTLS/HMAC/IP allowlist，超时与熔断保护，确保 <1s。

API 设计（对 PSP）

POST /v1/payments/authorize

```
{
 "psp_txn_id": "string",        // PSP 唯一交易号（做幂等）
 "shopper_id": 123,
 "amount": 300.00,
 "currency": "USD",
 "merchant": { "name": "Target", "address": "123 Main St" },
 "card_token": "opaque",        // 只接收 token，不碰 PAN
 "timestamp": "2025-08-27T00:00:00Z",
 "idempotency_key": "optional"   // 也可来自 Idempotency-Key header
}
```

Response 200（批准）

```
{
 "decision": "APPROVED",
 "approved_amount": 300.00,
 "order_id": "ord_abc",
 "reason_code": "OK",
 "psp_txn_id": "..."
}
```

Response 402（拒绝）

```
{
```

"decision": "DECLINED",

"reason_code": "AMOUNT_EXCEEDS_LIMIT|MERCHANT_NOT_ALLOWED|DUPLICATE|TIME_WINDOW|SYSTEM_UNAVAILABLE"

}

幂等语义：psp_txn_id（或 header 的 Idempotency-Key）在服务端 **唯一约束**。重复请求直接返回第一次决策。

数据模型（核心表 & Redis 快照）

PostgreSQL（强一致 & 审计）

- orders：order_id(PK), shopper_id, allowed_merchant_ids/name+addr 标准化, expected_total, tolerance_pct/by_category, time_window, status, spent_actual, created_at, updated_at
- payment_attempts：psp_txn_id(UNIQUE), order_id, shopper_id, amount, merchant_fingerprint, decision, reason, approved_at, raw_payload(JSONB)（审计/追踪）
- shoppers / merchants：基本档案与标准化信息（可拆服务）

Redis（L2，写通 / write-through）

- Key：active_order:{shopper_id}
- Value（JSON）：{ order_id, allowed_merchant_fp_set, expected_total, tolerance_rules, time_window, spent_so_far, partial_allowed(bool) }
- 只存"**每个 shopper 最新一张活跃订单**"→ 命中热路径；**写通策略**：订单创建/指派/修改时写 DB 同步写 Redis；核验通过时**原子地**回写 spent_so_far。

L1：服务节点内存短 TTL（几十毫秒 ~ 数秒）；L2：Redis；**cache miss** 再去 DB。

决策流程（<1s 热路径）

1. **认证鉴权**：mTLS + HMAC 签名校验 + IP allowlist（<20ms）。

2. **读快照**：L1/Redis 取 active_order:{shopper_id}（~1–5ms 命中，miss 回源 DB 20–40ms）。
3. **幂等检查**：查 payment_attempts by psp_txn_id（Postgres UNIQUE 或 Redis 布隆/集合快速挡回）。
4. **规则校验**（Decision Engine）：
    - **时间窗**：交易时间必须处于订单有效期。
    - **商户**：merchant 标准化后指纹匹配到 allowlist（或门店地理围栏匹配）。
    - **金额**：spent_so_far + amount <= expected_total * (1 + tolerance)；
        - 生鲜/称重品可有更高 **category tolerance**（比如 10–15%）；税费差、价格滞后容忍。
    - **分笔**：若 partial_allowed==true，允许多笔，**但总额不得超出上限**。
5. **原子更新 + 落审计**（二选一，二者都可答）
    - **Postgres 短事务**：SELECT ... FOR UPDATE 锁 orders 行，校验额度，INSERT payment_attempts（psp_txn_id 唯一）、UPDATE orders.spent_so_far，**提交**。
    - **Redis Lua 脚本**：对 active_order:{shopper_id} 做“读-判-加”的原子操作并写回；随后**异步**落库（或用事务消息/双写保障一致）。
6. **返回决策**：200/402，同时异步发事件（Kafka/SNS）给：风控学习、对账、通知、数据仓库。

超时守则：整个链路设置 300–600ms budget；cache 命中无 DB 的情况下 50ms 内完成；任何回源/重试都要带**短超时**和**熔断**；**超过阈值默认拒绝**（风险偏保守）。

安全（PCI/DSS 合规要点）

- 与 PSP 全链路 **mTLS**；请求体 **HMAC-SHA256** 签名（clock-skew 容忍）。
- **最小化持卡数据**：只接收 **token**，不落 PAN；所有敏感字段加密（KMS）与脱敏日志。
- **RBAC/ABAC**、机密管理（Vault/KMS）、审计日志**不可篡改**（WORM/append-only）。
- **速率限制**、WAF、重放保护（psp_txn_id + 时间窗 + nonce）。

- 数据保留策略与隐私合规（GDPR/CCPA）。

性能 & 可用性

- **水平扩展**：无状态服务 + **多 AZ** + **LB**（NLB/ALB）。
- **只读副本**：报告/风控离线查询用 read replica，不打扰热路径。
- **Redis 高可用**：多分片 + 哨兵/托管版，内置监控与自动故障转移。
- **Postgres**：主从、同步/异步复制，表分区（payment_attempts 按月分区），热点索引：
  - payment_attempts(psp_txn_id UNIQUE)
  - orders(shopper_id, status) 覆盖索引
  - orders(order_id) PK，必要时 orders(status, updated_at)
- **观测性**：指标（P50/P95/P99、命中率、拒绝原因分布）、日志、分布式追踪、熔断/重试告警。

DB 选型与扩展

- **起步**：**PostgreSQL**（强一致、事务、JSONB 审计很友好）。
- **扩展路径**：
  - 水平分库（按 shopper_id/region）或采用 **Aurora/Spanner** 这类分布式数据库。
  - payment_attempts 做时间分区 + 归档到对象存储。
  - **缓存前置**：热路径只依赖 Redis；DB 只承担幂等插入与审计更新。

幂等 & 并发控制（“刷两次只扣一次”）

- psp_txn_id **全局唯一约束**（DB UNIQUE 或 Redis SETNX 先占位）。
- 并发多笔：对 orders 行 SELECT ... FOR UPDATE 或 Redis Lua 脚本做 **原子加和** 并校验上限。

- 不同金额的重复请求：同 psp_txn_id → **直接返回首个决策**；不同 psp_txn_id 但同订单**累计额度**受控。

误差/风控策略（面试官爱问）

- **金额与下单价不一致**：
  - 价格时滞 & 称重误差 → **按品类容忍度**（如生鲜 10–15%，非生鲜 2–5%），或者 **期望价 ± 固定/比例阈值**。
  - **历史数据**自学习阈值（异步风控服务）。
- **非合作商户**：merchant 需匹配 allowlist（名称+地址标准化/地理围栏）；不匹配 → 拒绝。
- **分多次结账**：开启 partial_allowed，**累计不可超上限**（见并发控制）。
- **缺货/多买**：由订单服务变更 expected_total 或标记替代品；我们只核验**预算与商户**。
- **同时收到不同金额**：按**先到先审 + 累加上限** 验证，超限拒绝并告警。

容错 & 稳定性

- PSP 请求必须 **重试安全**（幂等）。
- 我方不可用/超时：**默认拒绝**并记录 SYSTEM_UNAVAILABLE（可配置）。
- 服务降级：cache 不可用 → 快速回源一次 DB；DB 慢 → 熔断拒绝；事件异步丢到本地队列/磁盘缓冲重放。

快速回答一些追问

- **为何用 cache，存什么，策略?**
  存“**shopper 的当前订单快照**”（允许商户、预算、容忍度、已花费…）；**写通**（order 指派/修改→DB 与 Redis 同写），核验通过后**原子更新 spent_so_far**；TTL 短、失效即回源。

- **如何优化读取**：覆盖索引、读副本仅用于报表、热路径只读 Redis；DB 的热 key 通过分区/哈希散列避免热点。
- **事务/回滚**：短事务包住 payment_attempts 插入与 orders.spent_so_far 更新；任何失败回滚并返回 402。
- **如何 scale DB**：分区、读写分离、分片或托管分布式 DB；payment_attempts 归档到对象存储 + 外表查询。
- **如何确定 merchant**：标准化（名称清洗、地址规范化、经纬度近邻）；同时存**门店 ID** 与**地理围栏**双重校验更稳。

以下是我聊到或者面试官问到的点，大家可以参考着准备

physical infrastructure: server, storage, network

data stores: SQL vs NoSQL

data model: merchant, shopper, order, transaction (includes order id in transaction table)

security: API token, pre-shared secret

performance considerations: load balancer, data partition, write through cache of order table

Monitoring: how to ensure the system performs expected?

Testing & Deployment: Load testing etc

Research & Analytics: how can data scientists leverage data for research?

尤其有两个特匪夷所思的点：

如果 database down 了，如何确保在 sla 一秒内 respond？这时直接 approve，因为他们 trust shoppers，认为 fraud rate 低

database 不需要 replica，面试官认为这个只会增加 latency 或者 inconsistency，不值得有这个 complexity