**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。为我们的一些零售合作伙伴建立‘暗店’（dark stores）。这些是专门的门店，备有全品类的杂货，不对公众开放，只供 Instacart 的购物员使用。**

**核心功能要求**

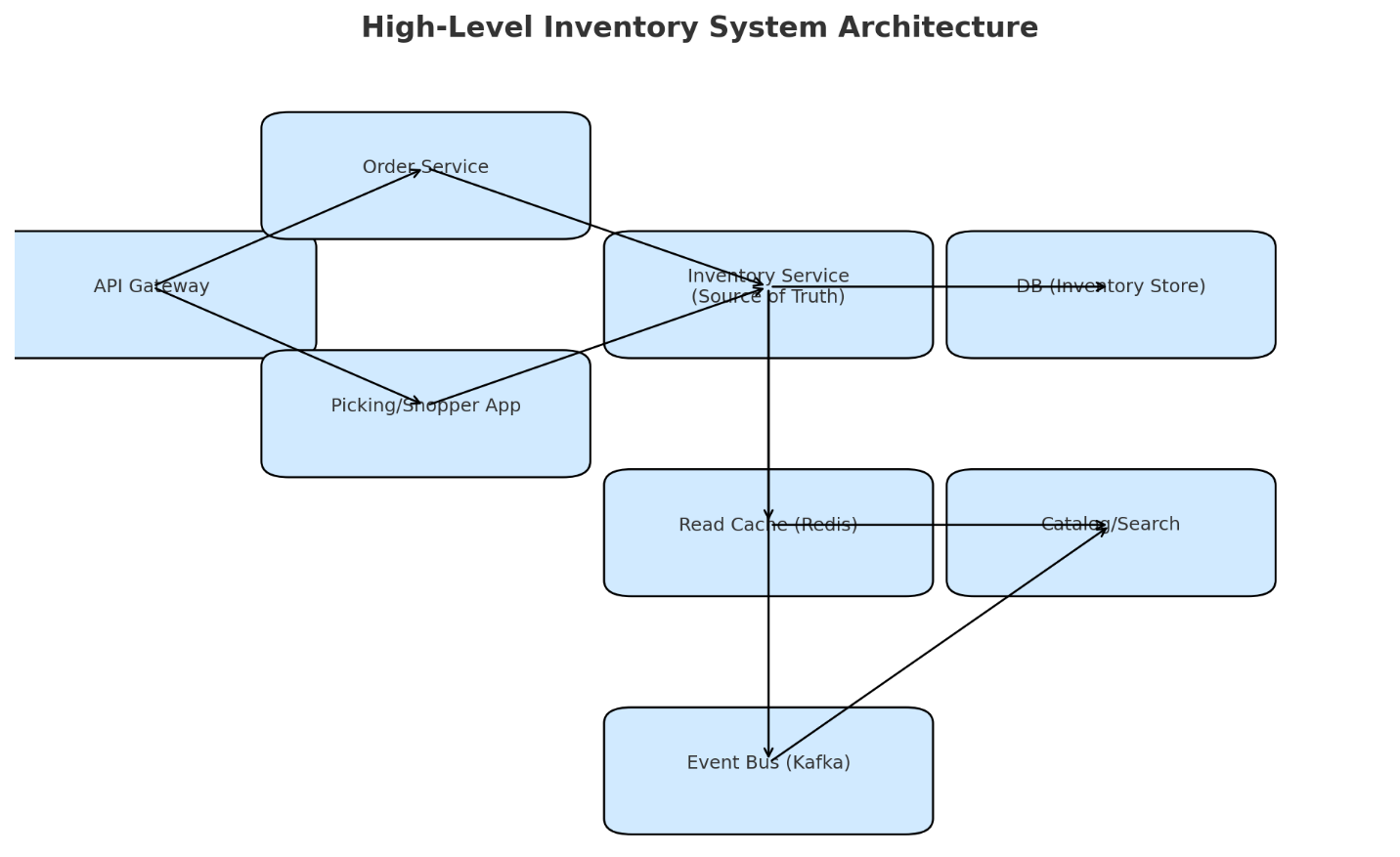
1. **Add stock（增加库存）**
   * **支持补货或增加仓库库存数量。**
2. **Reserve stock（预留库存）**
   * **当用户下单时，将库存临时锁定，防止其他订单抢占（防止超卖）。**
3. **Fulfill stock（完成订单）**
   * **当订单完成支付或发货时，真正扣减库存。**
4. **Clear reservation（清理预留）**
   * **订单取消或超时未支付时，将预留库存释放回可用库存。**

**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**



**3) 数据模型（最小可行） / Data Model (Minimal viable)**

**Inventory（每行=一个门店×一个SKU）**

**字段 / Columns**

| **字段** | **类型** | **约束** | **作用（ZH / EN）** |
| --- | --- | --- | --- |
| store\_id | BIGINT | PK(1/2) | 门店ID / Store identifier |
| sku\_id | BIGINT | PK(2/2) | 商品SKU / Product SKU |
| on\_hand | INT | NOT NULL, DEFAULT 0 | 实物在库 / Physical stock on hand |
| reserved | INT | NOT NULL, DEFAULT 0 | 已预留量 / Reserved for orders |
| version | BIGINT | NOT NULL, DEFAULT 0 | 乐观并发版本 / Optimistic concurrency version |
| updated\_at | TIMESTAMP | NOT NULL | 最近更新时间 / Last update timestamp |

**索引与约束 / Indexes & Constraints**

* PK：PRIMARY KEY (store\_id, sku\_id)
* 非负检查 / Non-negative checks：CHECK (on\_hand >= 0 AND reserved >= 0)
* 可用量不为负 / Availability invariant：CHECK (on\_hand - reserved >= 0)
* 热点读索引（可选）：INDEX inv\_store\_sku (store\_id, sku\_id)（PK 已覆盖）
* 分区建议 / Partitioning: **by store\_id** 或 **hash(store\_id, sku\_id)**

**Reservation（预留记录）**

**字段 / Columns**

| **字段** | **类型** | **约束** | **作用（ZH / EN）** |
| --- | --- | --- | --- |
| reservation\_id | UUID | PK | 预留ID / Reservation ID |
| store\_id | BIGINT | NOT NULL | 门店ID / Store ID |
| sku\_id | BIGINT | NOT NULL | SKU |
| user\_id | BIGINT | NOT NULL | 用户ID / Customer ID |
| order\_id | BIGINT | NULL | 订单ID（下单后可回填）/ Order ID (nullable) |
| qty | INT | NOT NULL, CHECK(qty > 0) | 预留数量 / Quantity reserved |
| status | SMALLINT | NOT NULL | 状态枚举 / Status enum（ACTIVE=1, RELEASED=2, FULFILLED=3, EXPIRED=4） |
| expires\_at | TIMESTAMP | NOT NULL | 过期时间 / TTL deadline |
| idempotency\_key | TEXT | UNIQUE | 幂等键 / Idempotency key |
| created\_at | TIMESTAMP | NOT NULL | 创建时间 / Created at |
| updated\_at | TIMESTAMP | NOT NULL | 更新时间 / Updated at |

**索引与约束 / Indexes & Constraints**

* PK：PRIMARY KEY (reservation\_id)
* 唯一：UNIQUE (idempotency\_key)
* 常用查询索引：
  + INDEX res\_store\_sku\_status (store\_id, sku\_id, status)（按门店+SKU+状态查活动预留）
  + INDEX res\_user\_active (user\_id, status)（查用户购物车/订单的预留）
  + INDEX res\_expire (status, expires\_at)（TTL 清理作业）
* 可选外键 / Optional FKs（视吞吐取舍）：(store\_id, sku\_id) → Inventory；order\_id → Orders

**PostgreSQL 示例 DDL（带注释 / With comments）**

-- INVENTORY

CREATE TABLE inventory (

store\_id BIGINT NOT NULL,

sku\_id BIGINT NOT NULL,

on\_hand INT NOT NULL DEFAULT 0,

reserved INT NOT NULL DEFAULT 0,

version BIGINT NOT NULL DEFAULT 0,

updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),

CONSTRAINT pk\_inventory PRIMARY KEY (store\_id, sku\_id),

CONSTRAINT chk\_nonneg CHECK (on\_hand >= 0 AND reserved >= 0),

CONSTRAINT chk\_available CHECK (on\_hand - reserved >= 0)

);

COMMENT ON TABLE inventory IS '每门店×SKU一行；DB为真相源 / One row per (store, sku); DB is SoT';

COMMENT ON COLUMN inventory.version IS '乐观并发 / Optimistic concurrency';

-- RESERVATION

CREATE TABLE reservation (

reservation\_id UUID PRIMARY KEY,

store\_id BIGINT NOT NULL,

sku\_id BIGINT NOT NULL,

user\_id BIGINT NOT NULL,

order\_id BIGINT NULL,

qty INT NOT NULL,

status SMALLINT NOT NULL, -- 1=ACTIVE,2=RELEASED,3=FULFILLED,4=EXPIRED

expires\_at TIMESTAMP NOT NULL,

idempotency\_key TEXT NOT NULL UNIQUE,

created\_at TIMESTAMP NOT NULL DEFAULT NOW(),

updated\_at TIMESTAMP NOT NULL DEFAULT NOW(),

CONSTRAINT chk\_qty\_pos CHECK (qty > 0)

-- 可按需开启外键（高并发下可只做应用校验）

-- ,CONSTRAINT fk\_res\_inventory FOREIGN KEY (store\_id, sku\_id) REFERENCES inventory(store\_id, sku\_id)

);

CREATE INDEX idx\_res\_store\_sku\_status ON reservation (store\_id, sku\_id, status);

CREATE INDEX idx\_res\_user\_status ON reservation (user\_id, status);

CREATE INDEX idx\_res\_expire ON reservation (status, expires\_at);

小贴士（面试可说）

* **Status 用 SMALLINT** 代替数据库 ENUM，迁移更轻松（应用层定义枚举映射）。
* **约束前置在 DB**：CHECK (on\_hand - reserved >= 0) 让不变量在存储层也被强制。
* **TTL 清理**：idx\_res\_expire(status, expires\_at) 让 Batch/Job 扫描只走索引区间。
* **分区策略**：Postgres 可 PARTITION BY HASH(store\_id)；MySQL 可用分库分表中间件。
* **审计表**：生产可增加 inventory\_audit（谁/何时/变更量/原因），满足追溯与合规。

**一眼可读的对照总结 / Quick Visual Summary**

**Inventory**

* Key: (store\_id, sku\_id)
* Numbers: on\_hand, reserved, version
* Rule: on\_hand - reserved >= 0

**Reservation**

* Key: reservation\_id, idempotency\_key (unique)
* Links: store\_id, sku\_id, user\_id, order\_id?
* Lifecycle: status, expires\_at
* Query Paths: (store, sku, status), (user, status), (status, expires\_at)
* **解释 / Explanation**
  + **ZH:** Fulfill → reserved 减少，同时 on\_hand 真正扣减。
  + **EN:** Fulfill → reserved decreases, on\_hand is actually deducted.
  + **ZH:** Clear → reserved 释放回 available。
  + **EN:** Clear → reserved is released back to available.

**4) 对外 API（核心四个） / Core APIs (4 endpoints)**

所有写操作都支持 Idempotency-Key，避免重复请求。  
All write operations support Idempotency-Key to avoid duplicate requests.

1. **Add stock**

* ZH: 增加库存
* EN: Add stock

POST /inventory/stock

Body: { store\_id, sku\_id, qty\_delta, reason }

Resp: { on\_hand, reserved, available }

1. **Reserve stock**

* ZH: 预留库存
* EN: Reserve stock

POST /reservations

Body: { store\_id, sku\_id, qty, user\_id, ttl\_sec }

Resp: { reservation\_id, expires\_at, available\_after }

Err: 409 (OUT\_OF\_STOCK)

1. **Fulfill stock**

* ZH: 完成订单扣减库存
* EN: Fulfill stock

POST /orders/{order\_id}/fulfill

Body: { reservations: [{reservation\_id, qty}], operator\_id }

Effect: reserved -= qty; on\_hand -= qty; reservation.status = FULFILLED

Resp: { fulfilled: true }

1. **Clear reservation**

* ZH: 清理预留库存
* EN: Clear reservation

POST /reservations/{reservation\_id}/clear

Effect: reserved -= qty; reservation.status = RELEASED

Resp: { cleared: true, available\_after }

（后台还需有 Expire Job，定期清理已过期的 ACTIVE Reservation，相当于自动 Clear。）  
(Background Expire Job required to clear expired ACTIVE reservations, equivalent to auto-Clear.)

**5) 关键读/写路径 / Key Read/Write Paths**

* **Add（补货/退货） / Add (Restock/Return)**
  + ZH: 在事务内 on\_hand += qty\_delta，保证 available ≥ 0；成功后发库存事件，刷新缓存。
  + EN: In a transaction, on\_hand += qty\_delta, ensuring available ≥ 0; upon success, emit event and refresh cache.
* **Reserve（预留） / Reserve**
  + ZH: 判断 available ≥ qty → 成功则 reserved += qty 并写 Reservation；否则返回 409。返回 reservation\_id + TTL。
  + EN: Check available ≥ qty → if success, reserved += qty and create Reservation; otherwise return 409. Returns reservation\_id + TTL.
* **Fulfill（完成） / Fulfill**
  + ZH: 支付成功后校验 Reservation 有效，再扣减 reserved & on\_hand，并设为 FULFILLED。
  + EN: After payment succeeds, validate Reservation is active, then deduct reserved & on\_hand, mark FULFILLED.
* **Clear（释放） / Clear**
  + ZH: 订单取消或超时，reserved -= qty，Reservation → RELEASED。
  + EN: On cancel or timeout, reserved -= qty, Reservation → RELEASED.

**6) 并发与正确性 / Concurrency & Correctness**

* **ZH:** 数据库为准：判断 + 更新必须在同一事务完成，避免“查后改”的竞态。
* **EN:** Database as source of truth: validation and update happen in the same transaction to avoid race conditions.
* **ZH:** 条件更新/乐观锁：只有 available ≥ qty 才允许增加 reserved。
* **EN:** Conditional update / optimistic locking: only allow increasing reserved if available ≥ qty.
* **ZH:** 幂等保证：Idempotency-Key 或 Reservation 唯一约束，确保重复请求不重复扣。
* **EN:** Idempotency guarantee: use Idempotency-Key or unique constraint on Reservation to prevent duplicate deductions.
* **一句话总结 / One-liner:**
  + **ZH:** “同一行 (store, sku) 做条件更新 + 本地事务，就能天然防超卖。”
  + **EN:** “Conditional updates + local transactions on the same (store, sku) row naturally prevent overselling.”

**7) 可扩展性 / Scalability**

* **ZH:** 读写分离：写入主库；读优先走 Redis（短 TTL/事件刷新），未命中回 DB。
* **EN:** Read/write split: writes go to master DB; reads use Redis (short TTL/event refresh), fallback to DB.
* **ZH:** 分区/分片：按 store\_id 或 hash(store\_id, sku\_id) 拆表/分库，避免热点。
* **EN:** Partition/sharding: split by store\_id or hash(store\_id, sku\_id) to avoid hotspots.
* **ZH:** 事件驱动：库存变化广播给搜索/风控/报表。
* **EN:** Event-driven: inventory changes broadcast to search, fraud detection, reporting.
* **ZH:** 降级策略：DB 压力大时临时只读可用量、暂停新预留；缓存失效时回源限流。
* **EN:** Degradation: under DB pressure, allow read-only availability, pause new reservations; fallback throttled reads when cache misses.

**8) 安全（高层） / Security (High-level)**

* **ZH:** 鉴权：OIDC / JWT；服务间 mTLS。
* **EN:** Authentication: OIDC / JWT; mTLS for service-to-service.
* **ZH:** 授权：用户仅能操作自己的 Reservation；内部接口走 RBAC。
* **EN:** Authorization: Users can only operate their own Reservations; RBAC for internal APIs.
* **ZH:** 审计：所有库存变更落日志，记录 who/what/when/why。
* **EN:** Auditing: log all inventory changes with who/what/when/why.
* **ZH:** 敏感数据：传输/存储加密。
* **EN:** Sensitive data: encryption in transit and at rest.

**9) 观测与上线 / Observability & Deployment**

* **SLO 示例 / Example SLOs:**
  + **ZH:** 查询可用量 P99 < 80ms；预留/清理/扣减 P99 < 200ms。
  + **EN:** Availability queries P99 < 80ms; reserve/clear/fulfill P99 < 200ms.
* **告警重点 / Key Alerts:**
  + **ZH:** available < 0（绝对不能出现）；Reservation 超时率；冲突重试率、P99 延迟。
  + **EN:** available < 0 (must never happen); reservation expiry rate; conflict retry rate; P99 latency.
* **上线实践 / Deployment Practices:**
  + **ZH:** 蓝绿/金丝雀发布，支持快速回滚；备份与恢复演练常态化。
  + **EN:** Blue-green/canary deployments with fast rollback; regular backup and recovery drills.

**Inventory System (Cheat Sheet) / 库存系统速记卡**

**0) 范围 / Scope**

* **ZH:** 多门店/暗店，SKU 粒度。目标：实时可用量、防超卖。DB = 真相源，缓存仅加速读。
* **EN:** Multi-store/dark stores, SKU granularity. Goal: real-time availability & prevent overselling. DB = source of truth; cache only for read optimization.

**1) 状态流转 / State Transitions**

Available --(Reserve)--> Reserved --(Fulfill)--> Shipped

Reserved --(Clear/Expire)--> Available

* **ZH:** available = on\_hand − reserved ≥ 0
* **EN:** available = on\_hand − reserved ≥ 0

**2) 组件 / Components**

* **ZH:** API Gateway（鉴权/限流）、Inventory Service（权威写入）、Order Service（下单/支付/发货）、Redis（加速读）、DB（真相源）、Kafka（事件广播）
* **EN:** API Gateway (auth/rate limiting), Inventory Service (authoritative writes), Order Service (orders/payment/shipping), Redis (read cache), DB (source of truth), Kafka (event broadcasting)

面试话术 / Interview line:

* **ZH:** 写进 Inventory+DB，读走缓存；DB 裁决。
* **EN:** All writes go to Inventory + DB; reads use cache; DB decides truth.

**3) 数据模型 / Data Model**

* **ZH:** Inventory：store\_id, sku\_id, on\_hand, reserved, version
* **EN:** Inventory: store\_id, sku\_id, on\_hand, reserved, version
* **ZH:** Reservation：id, store\_id, sku\_id, qty, status, expires\_at
* **EN:** Reservation: id, store\_id, sku\_id, qty, status, expires\_at

**4) 核心 API / Core APIs**

1. **Add** → on\_hand += q
2. **Reserve** → if available ≥ q, reserved += q (returns reservation\_id)
3. **Fulfill** → reserved -= q; on\_hand -= q; status=FULFILLED
4. **Clear** → reserved -= q; status=RELEASED

**5) 关键路径 / Key Flows**

* **ZH:** Add：入库更新 + 发事件
* **EN:** Add: Restock update + emit event
* **ZH:** Reserve：条件更新，生成 Reservation
* **EN:** Reserve: Conditional update, create Reservation
* **ZH:** Fulfill：校验 Reservation → 真正扣减
* **EN:** Fulfill: Validate Reservation → actual deduction
* **ZH:** Clear：释放 Reservation → 回 available
* **EN:** Clear: Release Reservation → back to available

**6) 并发与正确性 / Concurrency & Correctness**

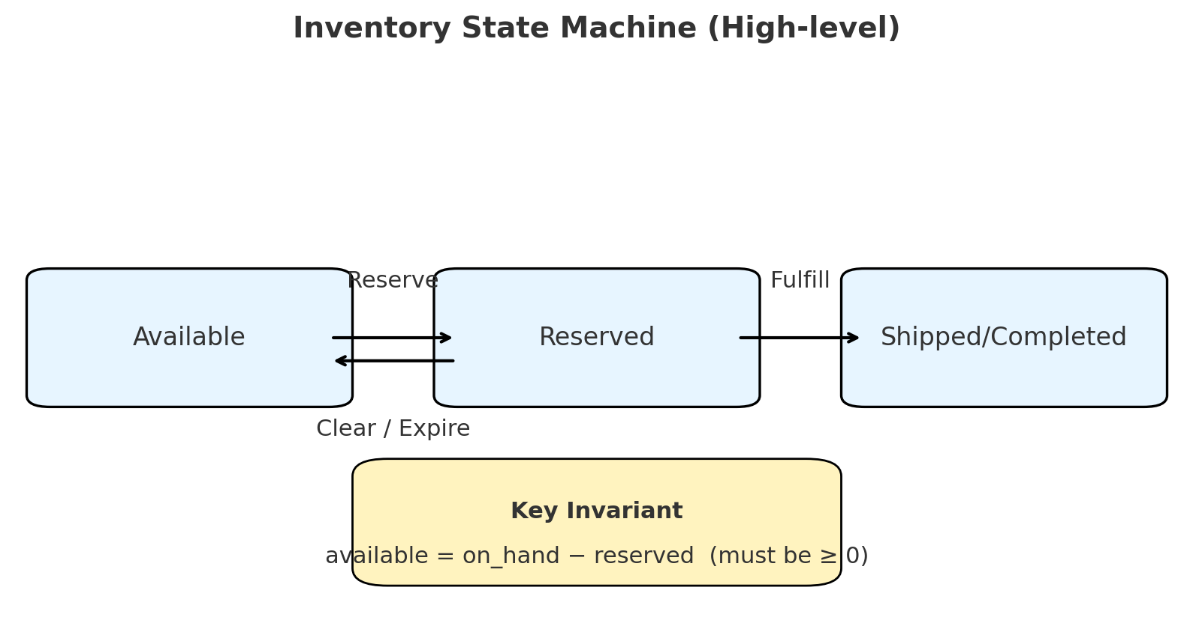
* **ZH:** 条件更新 + 本地事务，避免超卖；幂等键防重复；DB 为准，缓存最终一致。
* **EN:** Conditional update + local transaction to avoid overselling; idempotency key for dedupe; DB is truth, cache eventual consistency.

**7) 扩展性 / Scalability**

* **ZH:** 写主库，读缓存；TTL/事件刷新；按 store\_id 或 sku hash 分库；事件驱动扩展；降级只读。
* **EN:** Write to master, read from cache; TTL/event refresh; shard by store\_id or sku hash; event-driven expansion; degrade to read-only.

**8) 安全 & 上线 / Security & Deployment**

* **ZH:** 鉴权 OIDC/JWT, mTLS；授权 RBAC；审计 who/what/when/why；蓝绿/金丝雀发布；备份演练；监控 available<0, 超时率, P99 延迟。
* **EN:** Auth OIDC/JWT, mTLS; RBAC authorization; audit who/what/when/why; blue-green/canary deployments; backups & drills; monitor available<0, expiry rate, P99 latency.



**1) 并发控制 / Concurrency & Race Conditions**

**Q1：多人同时下单，如何防止超卖？**

* **EN:** “I enforce atomic conditional updates in the database. For each (store, sku), the check available >= qty and the increment of reserved happen in the same transaction. If two users try at the same time, only one succeeds, the other gets a 409 conflict. The database is the source of truth; cache is only for reads.”
* **ZH:** “我在数据库里做**条件更新**，把判断和增加预留放在同一个事务里。只有当 available >= qty 时才会成功；两个用户同时下单，只会有一个成功，另一个返回 409。数据库是唯一真相源，缓存只加速读。”

**Q2：乐观锁还是悲观锁？**

* **EN:** “Most of the time I use optimistic concurrency (with version or conditional update). For specific cases like multiple shoppers trying to consume the same reservation, I may use short-term pessimistic locks (SELECT FOR UPDATE).”
* **ZH:** “大多数场景用**乐观并发控制**（version 或条件更新）。但像多个 shopper 并发消费同一 reservation 这种情况，我会用短事务的**悲观锁**（SELECT FOR UPDATE）。”

**Q3：要不要用 Redis 原子操作？**

* **EN:** “Redis can be used for ultra-low-latency checks with Lua scripts, but the final decision must always be in the database. Cache is for acceleration, not for correctness.”
* **ZH:** “Redis 可以用 Lua 脚本做低延迟扣减，但**最终裁决必须在数据库**。缓存只用于加速，不用于保证正确性。”

**2) 扩展性 / Scaling**

**Q1：读写都很高，怎么扩展？**

* **EN:** “I follow a CQRS approach: writes go to the Inventory DB, reads go to Redis. Updates are pushed to an event bus (Kafka) so search, recommendation, and analytics stay updated asynchronously.”
* **ZH:** “我采用 **CQRS**：写入走 Inventory DB（强一致），读取走 Redis 缓存。库存变化通过事件总线（Kafka）广播给搜索、推荐、报表等下游异步更新。”

**Q2：如何分库分表，热点怎么处理？**

* **EN:** “Partition by store\_id or hash(store\_id, sku). For very hot SKUs, we can use sharded counters or even a single-threaded queue per SKU to serialize updates.”
* **ZH:** “可以按 store\_id 或 (store\_id, sku\_id) hash 来分库分表。如果出现超级热点 SKU，可以用**分片计数**或给该 SKU 上一个**单分区串行队列**，保证顺序处理。”

**Q3：缓存一致性如何保证？**

* **EN:** “On writes, we publish an event to update or invalidate cache keys. We also use short TTLs as a safety net, and include version/timestamp to prevent stale writes overriding fresh ones.”
* **ZH:** “写操作时发布事件更新或失效缓存，缓存本身设短 TTL 兜底。同时带 version 或时间戳，防止旧数据覆盖新数据。”

**3) 安全性 / Security**

**Q1：如何防止非法库存操作？**

* **EN:** “We combine authentication, authorization, and auditing. Authentication with OIDC/JWT and mTLS; authorization with RBAC/ABAC (customers can only act on their reservations; shoppers only for their store); and auditing every stock change with who/what/when/why.”
* **ZH:** “通过**鉴权、授权、审计**三层：鉴权用 OIDC/JWT 和 mTLS；授权用 RBAC/ABAC（顾客只能操作自己的 reservation，shopper 只能操作自己的门店）；所有库存变更都记录 who/what/when/why。”

**Q2：其他安全实践？**

* **EN:** “Idempotency keys, request signing, rate limiting to prevent abuse, least-privilege access to DB/queues, and encryption for data in transit and at rest.”
* **ZH:** “使用幂等键、请求签名、防刷限流、最小权限（DB/队列凭据走密钥管理），以及传输/存储加密。”

**4) 生产部署最佳实践 / Production Deployment**

**Q1：如何保证高可用和发布安全？**

* **EN:** “Deploy with Kubernetes across multiple AZs, DB with replicas. Use blue/green or canary releases with feature flags for safe rollouts and fast rollback.”
* **ZH:** “K8s 多 AZ 部署，数据库主从；发布走**蓝绿或金丝雀**，配合特性开关，能快速回滚。”

**Q2：可观测性和 SLO？**

* **EN:** “Key SLOs: inventory query P99 < 80ms, reserve/fulfill P99 < 200ms, and available should never be negative. Monitor reservation expiry rate, retry/conflict rate, cache hit rate, and message queue lag.”
* **ZH:** “关键 SLO：查询 P99 < 80ms，预留/扣减 P99 < 200ms，**available 永不为负**。监控预留超时率、冲突重试率、缓存命中率、消息队列积压。”

**Q3：备份与容灾？**

* **EN:** “Daily full backups with binlogs for point-in-time recovery, cross-region replicas, and regular disaster recovery drills.”
* **ZH:** “每天全量备份 + binlog 实现时间点恢复，跨区域热备，定期演练容灾恢复。”

**Q4：消息投递语义？**

* **EN:** “At-least-once delivery. Consumers must be idempotent, using business keys or deduplication tables. Support replaying historical topics to rebuild downstream indexes.”
* **ZH:** “采用 **至少一次投递**；消费者要幂等（用业务键或去重表）。支持历史消息重放，用于修复下游索引或报表。”

**面试收尾总结 / Wrap-up**

* **EN:** “We centralize strong consistency in the Inventory Service with atomic conditional updates, use cache and event-driven architecture for scale, secure the APIs with auth and RBAC, and deploy with blue/green and full observability. That gives us correctness, scalability, and operational safety.”
* **ZH:** “我们把强一致集中在 Inventory Service（条件更新防超卖），通过缓存和事件驱动实现扩展性，用鉴权/RBAC 保障安全，用蓝绿发布和完备监控保证可运维性。这样系统既**正确**又**可扩展**，还能**安全上线**。”

So the system I’d design is meant to handle multiple stores, including dark stores, where each product is identified by a SKU. The core goal is to always keep track of real-time availability and prevent overselling, even under high concurrency. I treat the database as the single source of truth, while the cache is only for speeding up reads.

Conceptually, each (store, sku) has two numbers: how many items are on hand, and how many are reserved. The key invariant is that on\_hand minus reserved must always be non-negative. The state machine is simple: an item starts as Available, then can be Reserved when a customer adds it to their order; it becomes Fulfilled once the order is paid and the shopper picks it; or, if the order is canceled or times out, the reservation is Cleared and goes back to Available.

At a high level, the system is made of an API gateway in front, an Inventory Service as the authoritative writer, an Order Service that handles checkout and calls into Inventory for fulfill and clear, a Redis cache for fast reads of availability, a relational database for strong consistency, and optionally an event bus like Kafka to broadcast inventory changes to search, recommendation, or analytics. My principle is: all writes go through the Inventory Service and database; reads are served from cache, but the database always has the final say.

The data model is straightforward: an Inventory table with on\_hand, reserved, and a version field for optimistic concurrency; and a Reservation table with a UUID, user, quantity, status, and expiry time. That lets us support add stock, reserve, fulfill, and clear as simple API calls. For example, add stock increases on\_hand; reserve checks availability and increments reserved with an expiry; fulfill decreases both reserved and on\_hand; clear just releases the reserved back to available.

To prevent race conditions, I rely on conditional updates inside a single transaction—so the check and the update happen atomically in the database. That naturally prevents overselling. Every write also uses an idempotency key, so retries won’t double count. For scaling, I’d separate reads and writes: writes stay in the main database, reads go through Redis with short TTL or event-driven updates. If we see very high load, we can shard by store or hash of SKU, and use events to keep downstream systems in sync.

Finally, for production readiness I’d add authentication and RBAC to secure the APIs, audit logs for all inventory changes, blue-green or canary deployments for safe releases, backups and recovery drills, and monitoring to make sure things like “available less than zero” never happen.