**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**

**🎯 System Design 面试开场 5 问模板**

**1️⃣ 范围 / Scope**

* **ZH: “请问这个系统只针对暗店，还是要同时支持普通门店？”**
* **EN: “Is this system focused only on dark stores, or should it also support normal retail stores?”**

**👉 明确设计范围，避免过度或不足。**

**2️⃣ 功能需求 / Functional Requirements**

* **ZH: “库存预留是否需要 TTL？也就是用户下单但不付款时，系统要不要自动释放库存？”**
* **EN: “Do reservations need a TTL, so if a customer doesn’t pay, the system automatically releases the stock?”**

**👉 确认核心业务逻辑。**

**3️⃣ 非功能需求 / Non-functional Requirements**

* **ZH: “大概的规模和并发量是多少？读写比例是怎样的，比如查询是否远多于下单？”**
* **EN: “What’s the expected scale and concurrency? And what’s the read/write ratio—for example, are reads much higher than writes?”**

**👉 决定是否需要缓存、分库分表、事件驱动。**

**4️⃣ 一致性 vs 性能 / Consistency vs Performance**

* **ZH: “对一致性的要求是什么？必须强一致，还是允许一定的最终一致性？”**
* **EN: “What’s the consistency requirement? Do we need strong consistency, or is eventual consistency acceptable?”**

**👉 决定是否要用强事务，还是可以牺牲一点一致性换性能。**

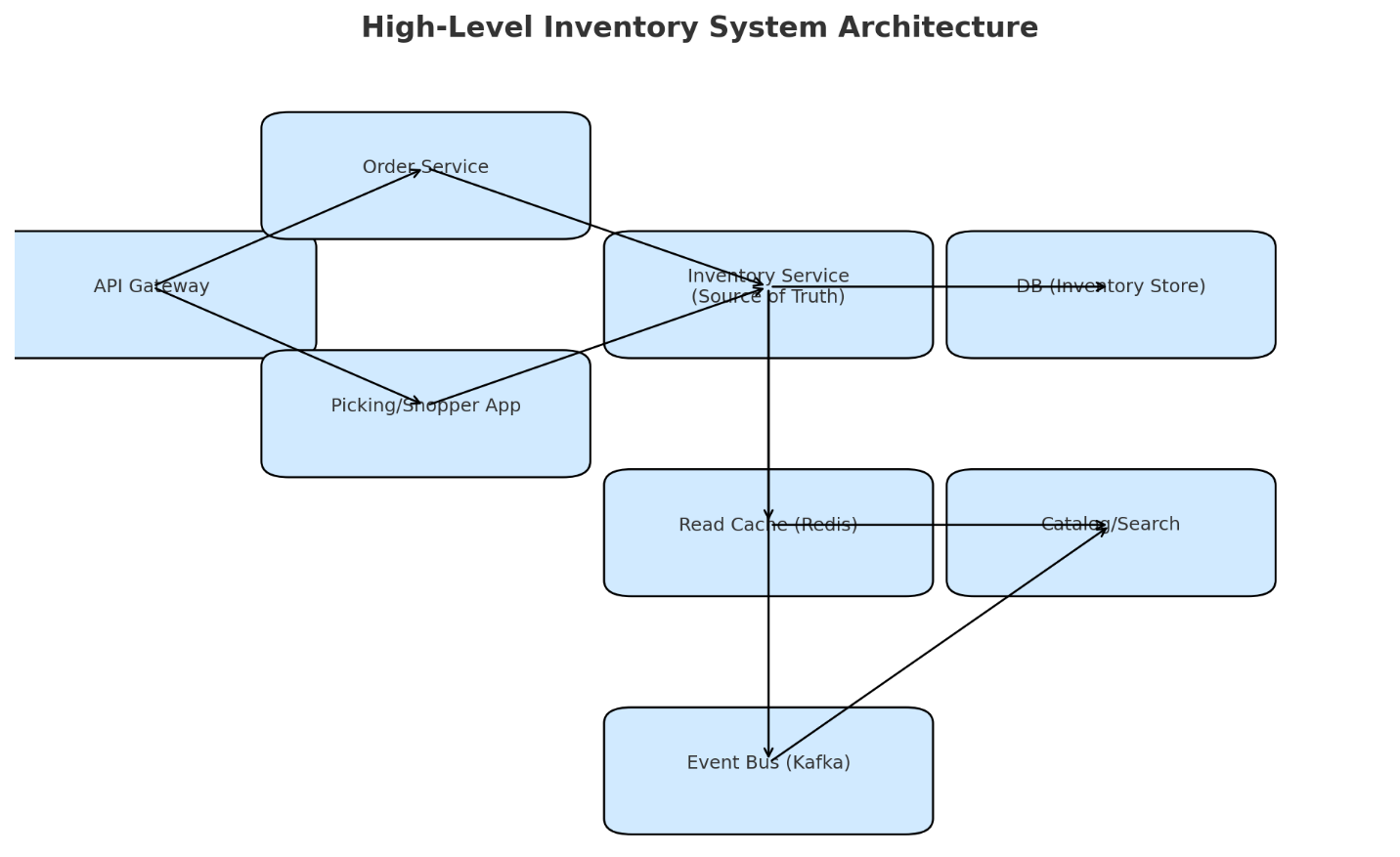
**5️⃣ 业务约束 & 安全 / Business Constraints & Security**

* **ZH: “谁可以调用这些 API？是顾客直接调用，还是都通过 order service？另外是否需要审计日志，记录每次库存变更的 who/what/when/why？”**
* **EN: “Who is allowed to call these APIs—do customers call directly, or only through the order service? And do we need full audit logs to record who/what/when/why for every inventory change?”**

**👉 确认鉴权、权限控制和合规性需求。**

**✨ 开场口播示例**

* **ZH:  
  “在开始设计之前，我想先澄清几个点。第一，这个系统只针对暗店，还是所有门店都要支持？第二，库存预留是否需要 TTL 自动清理？第三，预期的规模和并发量是多少，读写比例大概如何？第四，我们对一致性的要求是强一致还是可以最终一致？第五，谁能调用 API，是顾客还是只有订单服务？另外是否需要审计日志记录所有变更？这些信息明确了，我才能更好地做架构设计。”**
* **EN:  
  “Before diving into the design, I’d like to clarify a few things. First, is this system only for dark stores or all stores? Second, do reservations require a TTL for automatic expiry? Third, what’s the expected scale and read/write ratio? Fourth, do we need strong consistency or is eventual consistency acceptable? And fifth, who is authorized to call the APIs—customers directly or only via the order service—and do we need audit logs of all inventory changes? With these clarified, I can make better architectural trade-offs.”**



**Design: Inventory System (High-level) / 库存系统高层设计**

**0) 范围与假设 / Scope & Assumptions**

* **ZH:** 系统需要支持多门店/仓库（含暗店 Dark Store），每个商品以 SKU 唯一标识。
* **EN:** The system must support multiple stores/warehouses (including dark stores), with each product uniquely identified by a SKU.
* **ZH:** 实时可用量与防超卖是核心目标。
* **EN:** Real-time availability and preventing overselling are the core goals.
* **ZH:** 系统需要承受高并发的读写请求。
* **EN:** The system must handle high-concurrency read and write traffic.
* **ZH:** 数据库是库存的最终真相源（Source of Truth）；缓存只作为读优化。
* **EN:** The database is the source of truth for inventory; cache is only for read optimization.

**1) 核心概念与状态 / Core Concepts & States**

* **ZH:** 关键公式：available = on\_hand − reserved (必须始终 ≥ 0)
* **EN:** Key formula: available = on\_hand − reserved (must always be ≥ 0)
* **ZH:** 状态流转：
* Available --(Reserve)--> Reserved --(Fulfill)--> Shipped/Completed
* Reserved --(Clear/Expire)--> Available
* **EN:** State transitions:
* Available --(Reserve)--> Reserved --(Fulfill)--> Shipped/Completed
* Reserved --(Clear/Expire)--> Available

**2) 系统组件（High level） / System Components (High level)**

* **ZH:** API Gateway：负责鉴权、限流、审计。
* **EN:** API Gateway: Handles authentication, rate limiting, and auditing.
* **ZH:** Inventory Service：唯一的权威写入服务，处理 Add / Reserve / Fulfill / Clear。
* **EN:** Inventory Service: The single authoritative write service, handling Add / Reserve / Fulfill / Clear.
* **ZH:** Order Service：下单、支付、发货，调用 Inventory Service 扣减或清理库存。
* **EN:** Order Service: Manages order placement, payment, and shipping, calling Inventory Service for fulfillment or clearing.

Inventory Service is focused on stock itself—it’s the single source of truth for adds, reserves, fulfillments, and clears. Order Service manages the order lifecycle: placement, payment, cancellation. It never changes stock directly but calls Inventory Service to do so. This separation keeps responsibilities clear and ensures inventory consistency.

* **ZH:** Read Cache (Redis)：存储 (store, sku) 的可用量，加速查询。
* **EN:** Read Cache (Redis): Stores availability for (store, sku) to accelerate reads.
* **ZH:** DB (OLTP，如 Postgres/MySQL)：库存与预留的强一致存储。
* **EN:** DB (OLTP, e.g., Postgres/MySQL): Strongly consistent storage of inventory and reservations.
* **ZH:** Event Bus (Kafka, 可选)：将库存变更异步推送给搜索、推荐、风控等下游。
* **EN:** Event Bus (Kafka, optional): Asynchronously broadcasts inventory changes to search, recommendation, fraud detection, and other downstream systems.
* **面试口播要点：**
  + **ZH:** 写都进 Inventory Service + DB，读走缓存；DB 是最终裁决者。
  + **EN:** All writes go into Inventory Service + DB; reads use cache. The DB is the ultimate source of truth.

**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)**
* Version： The version column implements optimistic locking. Each update uses WHERE version = ?. The first request increments version from 0 to 1. When the second request arrives, it still expects version=0, but the row is now version=1, so the update fails. This way, concurrent updates don’t override each other and overselling is prevented

如果是悲观锁就用带 FOR UPDATE 的 SELECT：因为数据库在执行 SELECT … FOR UPDATE 时，会在行级别加上排他锁 (exclusive lock) 会锁住这行数据，直到当前事务 COMMIT 或 ROLLBACK， SELECT … FOR UPDATE is a pessimistic lock. It locks the selected rows so no other transaction can modify them until I commit. This way, concurrent updates to the same inventory row are serializedSELECT … FOR UPDATE is a pessimistic lock. It locks the selected rows so no other transaction can modify them until I commit. This way, concurrent updates to the same inventory row are serialized

**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) The idempotency key is a unique identifier attached to each API request. If the request is retried, the database constraint guarantees we don’t create duplicate reservations. We just return the existing one
* 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 }

ttl\_sec ：**Reserve stock API** 里，ttl\_sec 就是 **预留库存的过期时间（秒数）**

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 } operator\_id 是shopper

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: 分片的意思就是把数据拆到多个数据库里。默认可以按 store\_id 来拆，一个门店的数据放一个分片，这样查询效率高。如果某个门店很大或者某些 SKU 特别热门，就可以用 (store\_id, sku\_id) 的 hash 做更细的分片，把压力分散掉，避免热点。
* EN: Partitioning means splitting data across multiple databases. A simple approach is by store\_id, so all data for one store stays in one shard. That makes queries simple. But if one store is very large or some SKUs are extremely hot, we can shard further by hash(store\_id, sku\_id) to spread the load and 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: “事件驱动就是库存一旦变化，我们发一个事件到消息总线，然后下游系统像搜索、推荐、风控、报表去订阅这个事件，自己更新数据。这样可以让大家都实时知道库存变化，又不会给核心数据库增加压力。”

EN: “Event-driven means whenever inventory changes, we publish an event to a message bus, and downstream systems like search, recommendation, fraud detection, and reporting subscribe to it and update their data. This keeps everyone up to date without hammering the core database.”

* **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.

ZH: “降级的意思就是在系统快撑不住的时候，宁可功能打折扣，也不要全挂掉。比如数据库写压力太大时，可以暂停新的预留，只允许读库存；如果缓存失效，回源到数据库要做限流，不然数据库会被打爆。这样系统可以‘弯而不折’，在极端情况下还能保持核心功能可用。”

EN: “Degradation means that when the system is under stress, we’d rather reduce functionality than let it completely fail. For example, under DB pressure we can pause new reservations and allow only reads. If the cache fails, we throttle fallbacks to the DB to avoid overload. This way the system bends but doesn’t break, and core functionality stays available.”

**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.

**Production Deployment:**

**1. 高可用数据库 (High-availability DB)**

* **ZH:** 数据库是库存的真相源，所以必须做高可用。常见方案是 **主从复制 + 自动故障转移 (failover)**，甚至跨可用区多副本。这样一个节点宕机时，系统可以秒级切换到备用节点。
* **EN:** Since the DB is the source of truth, we must make it highly available—typically via master–replica with automatic failover, or multi-AZ replicas. This ensures the system survives node failures without losing availability.

**2. 负载均衡 (Load Balancing)**

* **ZH:** Inventory Service、API Gateway 等前端服务要有多个实例，流量通过 **负载均衡器 (LB)** 分发。这样即使某个实例挂了，请求还能自动切到健康的实例。
* **EN:** Services like Inventory or API Gateway run multiple instances behind a load balancer. If one goes down, traffic automatically shifts to healthy instances, ensuring zero-downtime.

**3. 日志与监控 (Logging & Monitoring)**

* **ZH:** 需要全链路的监控：
  + **指标监控 (Metrics)**：请求量、延迟、错误率、冲突重试率。
  + **日志 (Logs)**：每次库存变更记录 who/what/when/why。
  + **追踪 (Tracing)**：跨服务请求链路，用来排查性能瓶颈。
* **EN:** Full observability is key:
  + **Metrics** like request volume, latency, error/retry rates.
  + **Logs** for every stock change (who/what/when/why).
  + **Tracing** across services to debug bottlenecks.

**4. 备份与恢复 (Backups & Recovery)**

* ZH: 数据库必须有定期全量备份，并且要定期做 恢复演练。这样不仅能保证“有备份”，还能确认“能恢复”。在这里我们关注两个核心指标：
  + RPO (恢复点目标)：表示系统能容忍的数据丢失量，比如 RPO=5 分钟 → 最多丢最近 5 分钟的数据，所以要保证至少每 5 分钟可回放。
  + RTO (恢复时间目标)：表示系统从灾难发生到恢复可用的时间，比如 RTO=30 分钟 → 故障发生后半小时内必须恢复上线。  
    所以备份策略 + 恢复演练要确保这两个指标可以达标。
* EN: The database must take regular full backups, and we need regular recovery drills to prove not only that backups exist but also that we can restore. Two key metrics here are:
  + RPO (Recovery Point Objective): how much data we can afford to lose. For example, RPO=5 minutes means we can lose at most 5 minutes of data, so binlogs must allow replay within that window.
  + RTO (Recovery Time Objective): how quickly the system must be back online. For example, RTO=30 minutes means the system must be fully restored within half an hour.  
    Backup frequency and recovery procedures are designed to meet these targets.

**备份与恢复演练 (Backup & Recovery Drills)**

* **ZH:** 线上系统不仅要定期做数据库和关键数据的备份，还要定期做**恢复演练**。
  + 不仅验证“能备份”，更要确认“能从备份恢复”。
  + 这样一旦出现数据丢失或库损坏，能在目标 RPO/RTO 范围内恢复业务。
* **EN:** Production systems must not only take regular backups of databases and key data but also perform **recovery drills**.
  + It’s not enough to know backups exist—we must prove we can restore from them.
  + This ensures that if data loss or corruption occurs, we can recover within the agreed RPO/RTO.

**5. CI/CD 部署 (Continuous Integration & Continuous Deployment)**

* **ZH:** 部署流程需要自动化流水线：
  + **CI**：代码合并 → 自动跑单元测试、集成测试。
  + **CD**：自动部署到测试/预发/生产环境，配合蓝绿或金丝雀发布。
  + 出现问题时支持快速回滚。
* **EN:** Deployment must be automated with CI/CD:
  + **CI:** run unit/integration tests on every merge.
  + **CD:** deploy automatically to staging and production, with blue-green or canary strategies.
  + Must support fast rollback on failure.

**面试口播总结**

* **ZH:** “在生产部署时，我会确保几个关键点。首先是数据库要高可用，比如主从复制和自动故障转移；服务要跑在负载均衡器后面，多实例保证高可用。其次是全链路日志和监控，能及时发现问题。数据库要有备份并定期演练恢复，保证 RPO/RTO。最后是 CI/CD 自动化部署，结合蓝绿或金丝雀发布，出现问题可以快速回滚。”
* **EN:** “For production deployment, I’d ensure high availability on the DB with replicas and automatic failover, services behind load balancers with multiple instances, full logging and monitoring for observability, regular backups with recovery drills to meet RPO/RTO, and CI/CD automation with blue-green or canary deployments to enable safe rollouts and fast rollback.”

**6. 蓝绿发布 (Blue-Green Deployment)**

* **ZH:** 系统维护两套几乎一模一样的生产环境：**蓝环境**和**绿环境**。
  + 当前流量跑在“蓝”。
  + 新版本先部署到“绿”，测试验证没问题，再把流量切到“绿”。
  + 如果出问题，流量立刻切回“蓝”，实现快速回滚。
* **EN:** In blue-green deployment, we maintain two nearly identical environments: “blue” and “green.”
  + Traffic runs on blue.
  + We deploy the new version to green, test it, and then switch traffic over.
  + If issues arise, we can instantly roll back by switching back to blue.

**7. 金丝雀发布 (Canary Deployment)**

* **ZH:** 新版本不会一次性上线所有用户，而是先给一小部分用户（比如 1%）使用。
  + 如果指标正常，再逐步扩大比例到 10%、50%、100%。
  + 如果发现问题，只影响小部分用户，可以迅速回滚。
* **EN:** In a canary deployment, we release the new version to a small subset of users first, say 1%.
  + If metrics look good, we gradually increase to 10%, 50%, and eventually 100%.
  + If problems are detected, only a small group is affected, and we can roll back quickly.

**面试口播总结**

* **ZH:** “在上线实践上，我会用蓝绿或金丝雀发布来降低风险：蓝绿可以快速切换环境回滚，金丝雀可以先让少量用户试用再逐步放量。除此之外，还会定期做备份和恢复演练，确保不仅有备份，而且在出问题时能按 RPO/RTO 目标快速恢复。”
* **EN:** “For deployment practices, I’d use blue-green or canary deployments to reduce risk: blue-green lets us roll back instantly, while canary lets us release gradually. On top of that, I’d enforce regular backup and recovery drills, so we’re not only backing up but also verifying we can restore within our RPO/RTO targets.”

**what is RPO and RTO ?**

* **ZH:** “RPO 是能容忍丢多少数据，比如设成 5 分钟就表示最多只能丢最近 5 分钟的交易记录。RTO 是多快能恢复，比如设成 30 分钟就表示半小时内必须恢复服务。这两个指标决定了备份频率、容灾架构和演练目标。”
* **EN:** “RPO is about how much data we can afford to lose—for example, 5 minutes means at most 5 minutes of transactions are lost. RTO is about how fast we must recover—for example, 30 minutes means the system must be back online within half an hour. These two numbers drive our backup frequency, DR architecture, and recovery drills.”

**8.告警重点 / Key Alerts:**

**ZH:** available < 0（绝对不能出现）；Reservation 超时率；冲突重试率 Conflict retry rate、P99 延迟 P99 latency。

**EN:** available < 0 (must never happen); reservation expiry rate; conflict retry rate; P99 latency.

**P99 latency:** ZH: “P99 延迟就是 99% 的请求延迟都要小于某个值，只允许最慢的 1% 超过。比如 P99=200ms 就表示 99% 的请求小于 200 毫秒。它比平均值更能反映用户真实体验，尤其是尾部延迟。”

EN: “P99 latency means that 99% of requests finish faster than that threshold, and only the slowest 1% are worse. For example, P99=200ms means 99% of requests complete in under 200ms. It’s better than averages because it captures tail latency, which directly affects user experience.”

ZH: “告警上我会重点关注四个指标。第一，available 绝对不能小于 0，否则说明出现了超卖。第二，Reservation 的超时率，如果太高说明库存被无效锁定。第三，冲突重试率，如果过高说明有热点 SKU，需要调整策略。第四，P99 延迟，如果超标说明系统性能劣化，需要扩容或优化。”

EN: “For alerts, I focus on four things. First, available must never go below zero, otherwise overselling happened. Second, the reservation expiry rate—if it’s too high, stock is being locked unnecessarily. Third, the conflict retry rate—if it spikes, we may have hot SKUs and need to change strategy. And fourth, P99 latency—if it exceeds targets, performance is degrading and we need to scale or optimize.”

**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**

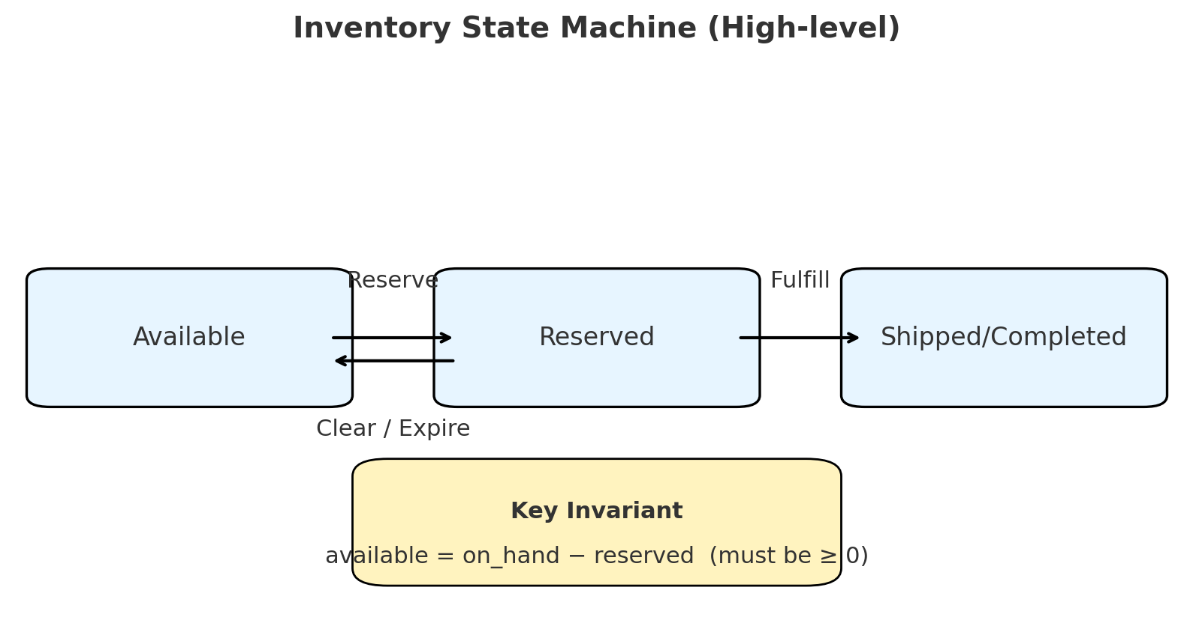
* **ZH:** 条件更新 + 本地事务，避免超卖；幂等键防重复；DB 为准，缓存最终一致。
* **EN:** Conditional update + local transaction to avoid overselling; idempotency key for dedupe; DB is truth, cache eventual consistency.

**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。”

ZH: “防止超卖不一定要显式加锁。我一般会用数据库的事务 + 条件更新，让‘判断是否有库存’和‘预留库存’在同一条 SQL 里完成。这样两个用户同时下单，只有一个能成功，另一个返回 409。如果遇到超级热点 SKU，可以考虑用悲观锁（SELECT FOR UPDATE），保证串行化修改。”

EN: “We don’t necessarily need explicit locks. I usually rely on atomic conditional updates inside a transaction, so checking availability and reserving stock happen in one step. If two users order at the same time, only one succeeds, the other gets a 409 conflict. For super-hot SKUs, we can switch to pessimistic locking with SELECT FOR UPDATE to serialize updates.”

**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）。”
* **乐观锁（Optimistic Locking）**
  + 不提前加锁，假设大多数情况下不会冲突。
  + 更新时用 version 字段（或条件更新）检测：
    - 如果 version 还是旧值 → 更新成功并 version+1。
    - 如果 version 已被别人改过 → 更新失败，说明有并发冲突，需要**重试或报错**。
  + 优点：吞吐量高，没有锁等待；缺点：冲突时要重试。
* **悲观锁（Pessimistic Locking）**
  + 更新之前就用 SELECT ... FOR UPDATE 锁住行。
  + 锁住后别人不能改，只能等这个事务结束。
  + 优点：不会发生写冲突；缺点：并发高时容易**排队等待或死锁**。

**面试口播总结**

* **ZH:** “对的。如果是乐观锁，我会用 version 来控制更新，发现冲突就重试；如果是悲观锁，我会在更新前用 SELECT FOR UPDATE 锁住那行数据，让其他事务必须等锁释放以后才能操作。”
* **EN:** “Exactly. With optimistic locking, I use a version field to control whether the update can succeed; if the version has changed, I know there was a conflict and I retry. With pessimistic locking, I lock the row upfront using SELECT FOR UPDATE, so other transactions must wait until I commit before they can update the same row.”
* **ZH:** “悲观锁的并发模型是：第一个事务先 SELECT ... FOR UPDATE 把行锁住，其他并发请求就会阻塞等待，等前一个事务提交后才能继续执行。如果不想阻塞，可以用 NOWAIT 让它立刻失败，或者用 SKIP LOCKED 跳过这行。在高并发下，我一般让普通 SKU 走乐观锁，热点 SKU 才用悲观锁。”
* **EN:** “With pessimistic locking, the first transaction locks the row, and all concurrent requests are blocked until it commits. That effectively serializes updates on the same row. If we don’t want blocking, we can use NOWAIT to fail immediately or SKIP LOCKED to skip locked rows. In practice, I’d use optimistic locking for most SKUs and switch to pessimistic only for very hot ones.”

**Q3：要不要用 Redis 原子操作？ Atomic operation means the whole operation happens in one indivisible step: either fully succeeds or not at all, with no interleaving.**

* **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.

**DB Schema**

So at the data model level, I keep it really simple with just two core tables: **Inventory** and **Reservation**.

The **Inventory** table has one row per store and SKU. It stores two key numbers: how many items are on hand and how many are already reserved. There’s also a version field for optimistic concurrency, and I enforce invariants at the database level—for example, both on\_hand and reserved must be non-negative, and on\_hand - reserved can never go below zero. That makes sure the system itself prevents overselling, even if there’s a bug in the application code.

Then the **Reservation** table tracks each hold when a customer adds items to their basket. It has a UUID reservation ID, the store and SKU, the user ID, quantity, a status field that goes from ACTIVE to either RELEASED, FULFILLED, or EXPIRED, and an expiry time so we can automatically clear holds that time out. Each reservation also carries an idempotency key, which ensures retries don’t create duplicates. I also add indexes to support the common queries: by store and SKU to check active reservations, by user to show their current basket, and by expiry time so a background job can efficiently release expired holds.

With just these two tables, we can implement all the key flows: add stock updates on\_hand, reserve increases reserved and creates a reservation record, fulfill consumes the reservation and deducts from on\_hand, and clear just releases the reservation back into available stock.

High level architecture + API design + DB Schema + Concurrency   
+ Scalability + Security + Deployment