Part 1: Code Review & Corrected Implementation

Problems in the Original Code

1. No validation of required or optional fields
2. SKU uniqueness not enforced
3. Multiple commits causing partial writes
4. No transactional safety
5. No error handling
6. Assumes a product belongs to only one warehouse
7. Price precision issues
8. Race conditions under concurrent requests

Correct Approach

1. Enforce business rules at the **database level**
2. Use **one atomic transaction**
3. Keep the API **thin and deterministic**
4. Let the database handle concurrency

Database Constraints

1. UNIQU (sku)
2. CHECK (price >= 0)
3. CHECK (quantity >= 0)
4. Composite key (product_id, warehouse_id) in inventory

Therefore my API approach is:

```
@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.json
    try:
        product = Product(
            name=data['name'],
            sku=data['sku'],
            price=Decimal(data['price'])
        )
```

```
    inventory = Inventory(
        product=product,
        warehouse_id=data['warehouse_id'],
        quantity=data.get('initial_quantity', 0)
    )
    db.session.add_all([product, inventory])
    db.session.commit()
    return {"product_id": product.id}, 201
    except IntegrityError:
    db.session.rollback()
    return {"error": "Invalid or duplicate data"}, 409
    except Exception:
    db.session.rollback()
    return {"error": "Internal server error"}, 500
```

Part 2: Database Design

1. Schema Design

The design focuses on **correctness, scalability, and clarity**, while keeping reads fast and writes safe.

Company:

```
Company (
   id              BIGINT PRIMARY KEY,
   name            VARCHAR(255) NOT NULL,
   created_at      TIMESTAMP NOT NULL DEFAULT NOW()
)
```

Warehouse:

```
Warehouse (
   id              BIGINT PRIMARY KEY,
   company_id      BIGINT NOT NULL REFERENCES Company(id),
   name            VARCHAR(255) NOT NULL,
   location        VARCHAR(255),
   created_at      TIMESTAMP NOT NULL DEFAULT NOW()
```

)

Relationship:

- One company → many warehouses

Indexes

- (company_id)

Product:

```
Product (
  id              BIGINT PRIMARY KEY,
  company_id      BIGINT NOT NULL REFERENCES Company(id),
  name            VARCHAR(255) NOT NULL,
  sku             VARCHAR(100) NOT NULL,
  price           DECIMAL(10,2) NOT NULL CHECK (price >= 0),
  created_at      TIMESTAMP NOT NULL DEFAULT NOW(),
  UNIQUE (sku)
)
```

Notes:

- SKU uniqueness enforced at DB level
- Price stored as DECIMAL to avoid precision issues

Indexes:

- (sku)
- (company_id)

Inventory (Current State):

```
Inventory (
  product_id      BIGINT REFERENCES Product(id),
  warehouse_id    BIGINT REFERENCES Warehouse(id),
  quantity        INT NOT NULL CHECK (quantity >= 0),
  updated_at      TIMESTAMP NOT NULL DEFAULT NOW(),
```

PRIMARY KEY (product_id, warehouse_id)

)

Purpose:

- Stores the **current quantity**
- Fast reads for APIs

Indexes:

- Composite primary key (product_id, warehouse_id)

  (warehouse_id)

InventoryEvent (Change Tracking):

InventoryEvent (

Id BIGINT PRIMARY KEY,

product_id BIGINT NOT NULL,

warehouse_id BIGINT NOT NULL,

quantity_change INT NOT NULL,

reason VARCHAR(100),

created_at TIMESTAMP NOT NULL DEFAULT NOW()

)

Purpose:

- Append-only history of inventory changes
- Used for audit, analytics, and debugging

Indexes

- (product_id, warehouse_id)
- (created_at)

Supplier:

Supplier (

id BIGINT PRIMARY KEY,

name VARCHAR(255) NOT NULL,

```
    contact_email   VARCHAR(255),
    created_at  TIMESTAMP NOT NULL DEFAULT NOW()
)
```

ProductSupplier:

```
ProductSupplier (
    product_id BIGINT REFERENCES Product(id),
    supplier_id BIGINT REFERENCES Supplier(id),
    PRIMARY KEY (product_id, supplier_id)
)
```

Relationship

- Many-to-many between products and suppliers

ProductBundle

```
ProductBundle (
    parent_product_id   BIGINT REFERENCES Product(id),
    child_product_id    BIGINT REFERENCES Product(id),
    quantity            INT NOT NULL CHECK (quantity > 0),
    PRIMARY KEY (parent_product_id, child_product_id)
)
```

Purpose

- Supports bundled products
- Allows nested or composite products

2. Missing Requirements / Questions for Product Team

These questions are necessary to finalize the design:

1. Is SKU **global** or **company-specific**?
2. Can a product have **multiple suppliers** simultaneously?
3. Should inventory updates be **real-time** or **eventually consistent**?
4. Do bundles have their **own SKU and price**, or are they derived?
5. Should inventory be **reserved** for pending orders?
6. How long should inventory history be retained?
7. Can warehouses transfer stock between each other?

8. Should low-stock thresholds vary per warehouse or per product?

3. Design Decisions & Justification

Separation of State and Events

- Inventory stores current quantity (fast reads)
- InventoryEvent stores history (audit and analytics)
- Prevents expensive recomputation

Database Constraints Over Application Logic

- UNIQUE, CHECK, and PRIMARY KEY constraints ensure correctness

  Prevents race conditions
- Simplifies API logic

Composite Keys for Inventory

- (product_id, warehouse_id) guarantees one row per product per warehouse
- Natural fit for multi-warehouse requirement

Indexes for Performance

- SKU lookups are O(log n)
- Inventory queries are fast even at scale
- Time-based indexes support reporting and alerts

Scalability

- Schema supports:
    1. Large companies
    2. Many warehouses
    3. High-frequency inventory updates
- Append-only events scale well under heavy writes

Part 3: Low-Stock Alerts API Implementation

Assumptions

To implement this correctly, the following assumptions are made:

1. **Low-stock threshold** is stored per product (or product type) as low_stock_threshold
2. **Recent sales activity** means at least one sale in the last **30 days**
3. A product can exist in **multiple warehouses**
4. A product can have **one primary supplier** for reordering

5. **Average daily sales** is precomputed and stored to avoid runtime aggregation
6. Inventory quantities are always non-negative

These assumptions are reasonable and necessary due to incomplete requirements.

Supporting Tables (Used by the API)

inventory

- product_id
- warehouse_id
- quantity

inventory_metrics

- product_id
- warehouse_id
- avg_daily_sales
- days_until_stockout
- last_sale_at

products

- id
- name
- sku
- low_stock_threshold

warehouses

- id
- company_id
- name

suppliers

- id
- name
- contact_email

product_supplier

- product_id
- supplier_id

API Endpoint:

Endpoint

GET /api/companies/{company_id}/alerts/low-stock

Implementation of the API:

```
@app.route('/api/companies/<int:company_id>/alerts/low-stock', methods=['GET'])
def get_low_stock_alerts(company_id):
    """
    Returns low-stock alerts for a company across all its warehouses.
    Only products with recent sales activity are considered.
    alerts = []
    query = """
        SELECT
            p.id AS product_id,
            p.name AS product_name,
            p.sku,
            w.id AS warehouse_id,
            w.name AS warehouse_name,
            i.quantity AS current_stock,
            p.low_stock_threshold,
            m.days_until_stockout,
            s.id AS supplier_id,
            s.name AS supplier_name,
            s.contact_email
        FROM inventory i
        JOIN inventory_metrics m
            ON i.product_id = m.product_id
           AND i.warehouse_id = m.warehouse_id
        JOIN products p ON i.product_id = p.id
        JOIN warehouses w ON i.warehouse_id = w.id
        JOIN product_supplier ps ON p.id = ps.product_id
        JOIN suppliers s ON ps.supplier_id = s.id
        WHERE
            w.company_id = :company_id
            AND i.quantity < p.low_stock_threshold
            AND m.last_sale_at >= NOW() - INTERVAL '30 days'
    """
```

```
    result = db.session.execute(query, {"company_id": company_id})

    for row in result:
        alerts.append({
            "product_id": row.product_id,
            "product_name": row.product_name,
            "sku": row.sku,
            "warehouse_id": row.warehouse_id,
            "warehouse_name": row.warehouse_name,
            "current_stock": row.current_stock,
            "threshold": row.low_stock_threshold,
            "days_until_stockout": row.days_until_stockout,
            "supplier": {
                "id": row.supplier_id,
                "name": row.supplier_name,
                "contact_email": row.contact_email
            }
        })
    return {
        "alerts": alerts,
        "total_alerts": len(alerts)
    }, 200
```

Why This Implementation Is Strong

1. Single Query Design

- One SQL query
- No N+1 problems
- Predictable performance

2. No Runtime Computation

- Sales metrics are precomputed
- No aggregation during request handling

- Fast response even at scale

3. Multi-Warehouse Support

- Inventory evaluated per (product, warehouse)
- Alerts generated independently per warehouse

4. Business Rules Clearly Enforced

- Low stock check
- Recent sales activity filter
- Supplier information included

Edge Cases Handled

1. **No alerts found**
    - Returns empty list with total_alerts = 0
2. **Product without recent sales**
    - Automatically excluded
3. **Multiple warehouses**
    - Each warehouse evaluated separately
4. **Division by zero risk**
    - Avoided by precomputing metrics offline
5. **Large data volume**
    - Query is index-friendly and cacheable

Indexing Recommendations

To ensure performance at scale:

inventory(product_id, warehouse_id)

inventory_metrics(product_id, warehouse_id)

inventory_metrics(last_sale_at)

warehouses(company_id)

products(low_stock_threshold)

Time Complexity

- Database query: **O(log n)** with indexes
- Application processing: **O(k)** where k = number of alerts

- No nested loops or repeated queries