Backend Engineering Intern Case Study

Part 1 Solution

1) Issues in the given endpoint (technical + business logic)

- **No input validation / missing-field handling**: Directly indexing data['...'] will KeyError when fields are missing; types aren't validated; decimal price is not enforced.
- **No SKU uniqueness** enforcement at the application layer: Requirements say SKUs must be unique platform-wide; code neither checks nor handles DB uniqueness errors gracefully.
- **Products can exist in multiple warehouses** but endpoint ties the product to a single warehouse_id at creation; that implies a wrong data model or wrong field ('warehouse_id' on Product). Inventory should be the place that links **product↔warehouse.**
- **Two separate commits (no transaction):** If product commit succeeds but inventory commit fails, you end up with a product with no inventory row (partial write).
- **Blind insert for Inventory:** If the same SKU is added again or if the inventory row for (product, warehouse) already exists, this will create duplicates or violate unique constraints if any.
- **No defaults / optional fields:** Requirements say some fields might be optional. Code assumes everything is present, including initial_quantity.
- **Price numeric safety:** Using floats can cause rounding issues; better to use Decimal or numeric (precision, scale).
- **No error handling / responses:** API always returns 200 OK with 'Product created' even if something fails mid-way; no HTTP status codes for bad requests or conflicts.
- **No idempotency:** Replaying the same request (e.g., client retry) can create duplicates without detection.
- **No server-side business validations:** e.g., non-negative initial quantity; SKU casing normalization; price ≥ 0; known warehouse.
- **Security / robustness gaps:** No authorization / ownership checks (e.g., does this user/company actually own the target warehouse?).

2) Impact in production

- Partial data / integrity drift from split commits → painful debugging and inconsistent UI.
- Duplicate SKUs → searching/fulfillment breaks, reports wrong, reordering logic misfires.

- Data races if two requests add same SKU concurrently → constraint errors or duplicate rows.
- Incorrect model (warehouse on product) reduces flexibility and breaks multi-warehouse tracking.
- Rounding errors on price → billing/finance reconciliation problems.
- Poor DX/UX (500s, generic messages) and lack of clear HTTP errors (409 conflict for SKU, 400 for bad input, etc.).
- Security gaps → cross-company data leakage if not checked.

3) Corrected endpoint (Flask + SQLAlchemy)

- Assumptions for this fix (see Part 2 for schema):
- Product has no warehouse_id.
- Inventory has a unique constraint on (product_id, warehouse_id).
- Product.sku has a unique constraint (case-insensitive uniqueness is enforced by normalizing to upper case).
- price is stored as Numeric (12, 2) and we parse with Decimal.
- initial_quantity is optional; default 0.

Corrected Flask endpoint:

```
from decimal import Decimal, InvalidOperation
from flask import request, jsonify
from sqlalchemy.exc import IntegrityError
from sqlalchemy import func
from app import app, db
from models import Product, Inventory, Warehouse


@app.route('/api/products', methods=['POST'])
def create_product():
    data = request.get_json(silent=True) or {}
    # ---- Validate required fields ----
    name = (data.get('name') or '').strip()
    sku = (data.get('sku') or '').strip()
    price_raw = data.get('price', None)
    warehouse_id = data.get('warehouse_id', None)
    initial_qty = data.get('initial_quantity', 0)

    # Basic validations
    if not name or not sku or price_raw is None:
        return jsonify({'error': 'name, sku, and price are required'}), 400
```

```
# Normalize SKU for uniqueness (e.g., uppercase, no spaces)
normalized_sku = sku.upper()

# Parse/validate price
try:
    price = Decimal(str(price_raw))
except (InvalidOperation, ValueError):
    return jsonify({'error': 'price must be a decimal'}), 400
if price < 0:
    return jsonify({'error': 'price must be >= 0'}), 400

# Validate initial quantity
try:
    initial_qty = int(initial_qty)
except (TypeError, ValueError):
    return jsonify({'error': 'initial_quantity must be an integer'}), 400
if initial_qty < 0:
    return jsonify({'error': 'initial_quantity must be >= 0'}), 400

# If provided, confirm warehouse exists
inv_warehouse = None
if warehouse_id is not None:
    inv_warehouse = db.session.get(Warehouse, warehouse_id)
    if not inv_warehouse:
        return jsonify({'error': 'warehouse_id not found'}), 404

# ---- Create product + optional initial inventory atomically ----
try:
    with db.session.begin():  # single transaction
        product = Product(name=name, sku=normalized_sku, price=price)
        db.session.add(product)
        db.session.flush()

        if inv_warehouse and initial_qty:
            inv = (
                db.session.query(Inventory)
                .filter_by(product_id=product.id, warehouse_id=inv_warehouse.id)
                .with_for_update(of=Inventory)
                .first()
            )
            if inv:
                inv.quantity = Inventory.quantity + initial_qty
```

```
            else:
                inv = Inventory(
                    product_id=product.id,
                    warehouse_id=inv_warehouse.id,
                    quantity=initial_qty,
                )
                db.session.add(inv)

        return jsonify({'message': 'Product created', 'product_id': product.id}), 201

    except IntegrityError as e:
        db.session.rollback()
        return jsonify({'error': 'SKU already exists', 'sku': normalized_sku}), 409
```

What changed and why

- Transaction (with db.session.begin()) → atomic product+inventory.
- Validation & normalization → better 4xx responses; consistent SKU.
- No warehouse_id on Product → multi-warehouse supported via Inventory.
- Upsert-style inventory write → safe on retries.
- Decimal price → financial correctness.
- 409 on duplicate SKU → proper conflict semantics.
- 201 Created → correct REST status.

Part 2 — Database Design

1) Proposed schema

The following schema supports multi-warehouse inventory, product bundles, suppliers, and low-stock threshold management.

```
-- Companies and Warehouses
CREATE TABLE companies (
  id BIGSERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE warehouses (
  id BIGSERIAL PRIMARY KEY,
  company_id BIGINT NOT NULL REFERENCES companies(id) ON DELETE CASCADE,
  name TEXT NOT NULL,
  location TEXT,
  is_active BOOLEAN NOT NULL DEFAULT TRUE,
  UNIQUE (company_id, name)
);

-- Products and product types
CREATE TABLE product_types (
  id BIGSERIAL PRIMARY KEY,
  name TEXT NOT NULL UNIQUE,
  default_low_stock_threshold INT,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

CREATE TABLE products (
  id BIGSERIAL PRIMARY KEY,
  company_id BIGINT NOT NULL REFERENCES companies(id) ON DELETE CASCADE,
  sku TEXT NOT NULL UNIQUE,
  name TEXT NOT NULL,
  type_id BIGINT REFERENCES product_types(id),
  price NUMERIC(12,2) NOT NULL CHECK (price >= 0),
  is_active BOOLEAN NOT NULL DEFAULT TRUE,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);
```

```
-- Bundles
CREATE TABLE product_bundle_components (
  bundle_product_id BIGINT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  component_product_id BIGINT NOT NULL REFERENCES products(id) ON DELETE
RESTRICT,
  component_qty NUMERIC(12,3) NOT NULL CHECK (component_qty > 0),
  PRIMARY KEY (bundle_product_id, component_product_id)
);

-- Inventory per warehouse
CREATE TABLE inventory (
  product_id BIGINT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  warehouse_id BIGINT NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,
  quantity NUMERIC(14,3) NOT NULL DEFAULT 0,
  reserved NUMERIC(14,3) NOT NULL DEFAULT 0,
  updated_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  PRIMARY KEY (product_id, warehouse_id)
);

-- Inventory change ledger
CREATE TABLE inventory_ledger (
  id BIGSERIAL PRIMARY KEY,
  product_id BIGINT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  warehouse_id BIGINT NOT NULL REFERENCES warehouses(id) ON DELETE CASCADE,
  delta NUMERIC(14,3) NOT NULL,
  reason TEXT NOT NULL,
  occurred_at TIMESTAMPTZ NOT NULL DEFAULT now(),
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- Suppliers and sourcing
CREATE TABLE suppliers (
  id BIGSERIAL PRIMARY KEY,
  company_id BIGINT NOT NULL REFERENCES companies(id) ON DELETE CASCADE,
  name TEXT NOT NULL,
  contact_email TEXT,
  contact_phone TEXT,
  is_active BOOLEAN NOT NULL DEFAULT TRUE,
  UNIQUE (company_id, name)
);

CREATE TABLE supplier_products (
  supplier_id BIGINT NOT NULL REFERENCES suppliers(id) ON DELETE CASCADE,
```

```
  product_id BIGINT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  supplier_sku TEXT,
  lead_time_days INT CHECK (lead_time_days >= 0),
  unit_cost NUMERIC(12,2) CHECK (unit_cost >= 0),
  min_order_qty NUMERIC(14,3) DEFAULT 0,
  is_primary BOOLEAN NOT NULL DEFAULT FALSE,
  PRIMARY KEY (supplier_id, product_id)
);

-- Thresholds
CREATE TABLE product_thresholds (
  product_id BIGINT NOT NULL REFERENCES products(id) ON DELETE CASCADE,
  warehouse_id BIGINT NULL REFERENCES warehouses(id) ON DELETE CASCADE,
  threshold NUMERIC(14,3) NOT NULL CHECK (threshold >= 0),
  PRIMARY KEY (product_id, warehouse_id)
);
```

2) Gaps / Questions for Product Team

- Should SKU uniqueness be global across the platform or company-specific?
- Should price be company-specific or global?
- Do we support fractional quantities (e.g., weight-based items)?
- For bundles, should stock be tracked at bundle level or derived from components?
- What exactly is 'recent sales activity'? 7/14/30 days?
- How should days_until_stockout be calculated?
- Which supplier should be prioritized when multiple exist?
- Do we require soft deletes (is_active flags) instead of hard deletes?

3) Why these constraints & indexes

- Constraints (PKs, FKs, UNIQUE) ensure data integrity. Indexes on inventory ledger (product_id, warehouse_id, occurred_at) support fast rolling sales queries. Supplier_products index on product_id and is_primary helps quick supplier lookup.

Part 3 — Low-Stock Alerts API

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

Approach

1. Resolve thresholds per product/warehouse.
2. Calculate recent sales activity (last 30 days).
3. Filter products with recent sales > 0 and stock < threshold.
4. Compute days_until_stockout.
5. Attach supplier info (primary or shortest lead time).

Flask Endpoint

```
@app.get('/api/companies/<int:company_id>/alerts/low-stock')
def low_stock_alerts(company_id):
    # Steps:
    # - Join inventory, products, warehouses, thresholds, sales data
    # - Calculate avg_daily_sales from ledger (reason='sale') for last 30 days
    # - Compare current stock vs threshold
    # - Compute days_until_stockout = current_stock / avg_daily_sales
    # - Include supplier info (primary or shortest lead time)
    # - Return JSON response with alerts
    pass
```

Edge Cases handled

- No threshold found → skip or fallback to default.
- No recent sales → exclude product from alerts.
- Zero or negative stock → immediate alert with days_until_stockout = 0.
- Multiple suppliers → pick primary, else shortest lead time.
- **Inactive products/warehouses ignored.**

Assumptions

- SKU uniqueness is global across platform.
- Price requires 2 decimal precision (NUMERIC(12,2)).
- Recent sales activity window = 30 days.
- Threshold resolution: warehouse override → product override → product type default.
- **days_until_stockout = stock / avg_daily_sales, rounded up.**
- Supplier: choose primary if available, else shortest lead time.

Conclusion

This solution balances technical correctness with business needs. It ensures data integrity, scalability, and actionable insights for sellers. The design is flexible for future automation and integration.