# Task Solution

---

## Task 1: Code Review & Debugging

### Problem :

1. No SKU uniqueness.
2. 2 different commits and price precision issue.

### Approach:

I focused on data integrity first. In production systems, partial writes are worse. So I enforced uniqueness, used precise numeric types, and wrapped creation in a single transaction. I also aligned the data model with the business rule that products can exist across multiple warehouses.

### Implementation:

```
@app.route("/api/products", methods=["POST"])
def create_product():
    data = request.get_json(force=True)

    name = data.get("name")
    sku = data.get("sku")
    price = Decimal(str(data.get("price")))
    warehouse_id = data.get("warehouse_id")
    initial_quantity = data.get("initial_quantity", 0)

    existing = db.session.execute(
        select(Product.id).where(Product.sku == sku)
    ).first()
    if existing:
        return jsonify({"error": "SKU already exists"}), 409

    product = Product(
        name=name,
        sku=sku,
        price=price
    )

    db.session.add(product)
    db.session.flush()

    inventory = Inventory(
        product_id=product.id,
        warehouse_id=warehouse_id,
        quantity=initial_quantity
    )

    db.session.add(inventory)
    db.session.commit()

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

### Key Steps:

1. SKU uniqueness validation before insert
2. Used session.flush() instead of early commit and used Decimal for price accuracy.
3. Single transaction : Both product & Inventory succeed or fail

---

## Task 2: Database Design

### Approach:

I started by identifying the core business entities [Company, Warehouse, Product, Supplier] and separating them based on responsibility to avoid data duplication and tight coupling. Since products can exist in multiple warehouses with different quantities, inventory was modeled as a relationship table rather than an attribute of the product itself.

### Implementation:

```
CREATE TABLE companies (
    id BIGSERIAL PRIMARY KEY,
    name TEXT NOT NULL
);

CREATE TABLE warehouses (
    id BIGSERIAL PRIMARY KEY,
    company_id BIGINT NOT NULL REFERENCES companies(id),
    name TEXT NOT NULL
);

CREATE TABLE suppliers (
    id BIGSERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    contact_email TEXT
);

CREATE TABLE products (
    id BIGSERIAL PRIMARY KEY,
    company_id BIGINT NOT NULL REFERENCES companies(id),
    name TEXT NOT NULL,
    sku TEXT NOT NULL UNIQUE,
    price NUMERIC(12,2) NOT NULL,
    product_type TEXT NOT NULL,
    supplier_id BIGINT REFERENCES suppliers(id)
);

CREATE TABLE inventories (
    id BIGSERIAL PRIMARY KEY,
    product_id BIGINT NOT NULL REFERENCES products(id),
    warehouse_id BIGINT NOT NULL REFERENCES warehouses(id),
    quantity INTEGER NOT NULL,
    UNIQUE (product_id, warehouse_id)
);

CREATE TABLE inventory_events (
    id BIGSERIAL PRIMARY KEY,
    product_id BIGINT NOT NULL REFERENCES products(id),
    warehouse_id BIGINT NOT NULL REFERENCES warehouses(id),
    delta INTEGER NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT now()
);

CREATE TABLE product_bundles (
    bundle_id BIGINT NOT NULL REFERENCES products(id),
    component_id BIGINT NOT NULL REFERENCES products(id),
    quantity INTEGER NOT NULL,
    PRIMARY KEY (bundle_id, component_id)
);

CREATE INDEX idx_inventory_product ON inventories(product_id);
CREATE INDEX idx_inventory_warehouse ON inventories(warehouse_id);
CREATE INDEX idx_inventory_events_time ON inventory_events(created_at);
```

### Key Steps:

1. Identified Core Entities and Ownership
2. Modeled Inventory as a Relationship
3. Integrated Supplier Management
4. Applied Indexes

---

## Task 3: API Implementation

### Objective

The objective was to design and implement an API endpoint that identifies and returns low-stock alerts for a given company. The endpoint needed to account for multiple warehouses, product-specific stock thresholds, recent sales activity, and supplier details required for reordering, while returning a structured and frontend-ready response.

### Approach:

The solution needed to aggregate data across multiple tables [products, warehouses, inventory, suppliers, and sales thresholds] while applying business rules dynamically.

A relational database with SQL-based aggregation was chosen because:

- The logic involves complex joins and calculations
- Performance and predictability are critical for alert generation
- SQL allows precise control over filtering, grouping, and derived fields such as stock-out estimates

### Implementation:

```
@app.route("/api/companies/<int:company_id>/alerts/low-stock", methods=["GET"])
```

```
def low_stock_alerts(company_id):
    query = text("""
        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,
            t.threshold,
            CEIL(i.quantity / NULLIF(t.daily_sales, 0)) AS days_until_stockout,
            s.id AS supplier_id,
            s.name AS supplier_name,
            s.contact_email
        FROM inventories i
        JOIN products p ON p.id = i.product_id
        JOIN warehouses w ON w.id = i.warehouse_id
        JOIN suppliers s ON s.id = p.supplier_id
        JOIN product_thresholds t ON t.product_type = p.product_type
        WHERE w.company_id = :company_id
          AND i.quantity < t.threshold
          AND t.daily_sales > 0
    """)

    rows = db.session.execute(query, {"company_id": company_id}).mappings().all()

    alerts = [
        {
            "product_id": r["product_id"],
            "product_name": r["product_name"],
            "sku": r["sku"],
            "warehouse_id": r["warehouse_id"],
            "warehouse_name": r["warehouse_name"],
            "current_stock": r["current_stock"],
            "threshold": r["threshold"],
            "days_until_stockout": r["days_until_stockout"],
            "supplier": {
                "id": r["supplier_id"],
                "name": r["supplier_name"],
                "contact_email": r["contact_email"]
            }
        }
        for r in rows
    ]
```

```
return jsonify(
    {
        "alerts": alerts,
        "total_alerts": len(alerts)
    }
)
```

## Key Steps:

1. Defined Business Filters
2. Joined Required Entities and calculated different matrics
3. Structured API Response