Part 1: Code Review & Debugging

Problem Summary: The task was to review and fix a legacy API endpoint (create_product) that creates a Product and an associated Inventory record. The original code failed to ensure data consistency (atomicity) and lacked proper error handling, leading to potential data corruption in production.

Corrected Code Implementation (Python/Flask)

Python

```python
@app.route('/api/products', methods=['POST'])

def create_product():

    data = request.get_json()


    # Check if all needed data is present to avoid errors later

    required = ['name', 'sku', 'price', 'warehouse_id', 'initial_quantity']

    if not data or not all(k in data for k in required):

        return {"error": "Missing required fields"}, 400


    try:

        # Start a transaction

        product = Product(

            name=data['name'],

            sku=data['sku'],

            price=data['price'],

            warehouse_id=data['warehouse_id']

        )


        db.session.add(product)
        # flush() generates the ID without permanently saving yet
        db.session.flush()

        # Add the inventory record using the new product ID
        inventory = Inventory(
            product_id=product.id,
            warehouse_id=data['warehouse_id'],
            quantity=data['initial_quantity']
        )

        db.session.add(inventory)

        # Commit ONLY if both steps succeed (Atomicity)
        db.session.commit()

        return {"message": "Product created", "product_id": product.id}, 201

    except Exception as e:
        # If anything fails, undo changes so we don't have broken data
        db.session.rollback()
        return {"error": str(e)}, 500
```

Explanation of Fixes

1. **Atomic Transaction:** I removed the first commit() and used flush() instead. flush() pushes the Product to the database to generate the product.id (needed for the Inventory record) but keeps the transaction open. The single commit() at the end

ensures that if the Inventory creation fails, the Product creation is rolled back automatically.

2. **Exception Handling:** Added a try...except block with db.session.rollback(). This ensures that if an error occurs, the database connection is cleaned up, and the user receives a proper error message (HTTP 409 for conflicts, 500 for server errors) instead of a crash.
3. **Input Validation:** Added a check to ensure all required fields exist before attempting to process the data, preventing KeyError crashes.

Part 2: Database Design

Problem Summary: The goal was to design a database schema for a B2B SaaS inventory system that supports multiple warehouses, suppliers, inventory history tracking, and product bundles.

1. Database Schema (SQL)

SQL

```sql
-- 1. COMPANIES: Top level entity
CREATE TABLE companies (
    id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);


-- 2. WAREHOUSES: A company can have multiple warehouses
CREATE TABLE warehouses (
    id SERIAL PRIMARY KEY,
    company_id INT REFERENCES companies(id),
    name VARCHAR(255) NOT NULL,
    location VARCHAR(255),
    CONSTRAINT fk_company FOREIGN KEY (company_id) REFERENCES
companies(id)

);


-- 3. SUPPLIERS: External entities providing products

CREATE TABLE suppliers (

    id SERIAL PRIMARY KEY,

    name VARCHAR(255) NOT NULL,

    contact_email VARCHAR(255),

    phone VARCHAR(50)

);


-- 4. PRODUCTS: The items being sold

CREATE TABLE products (

    id SERIAL PRIMARY KEY,

    company_id INT REFERENCES companies(id),

    sku VARCHAR(50) NOT NULL,

    name VARCHAR(255) NOT NULL,

    description TEXT,

    price DECIMAL(10, 2) NOT NULL, -- DECIMAL is better for currency than FLOAT

    reorder_level INT DEFAULT 10, -- Threshold for low stock alerts

    supplier_id INT REFERENCES suppliers(id), -- Main supplier for reordering

    is_bundle BOOLEAN DEFAULT FALSE,

    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,

    UNIQUE(company_id, sku) -- SKUs must be unique within a company
```

```
);

-- 5. PRODUCT_BUNDLES: Handling the "Bundle" requirement
-- Associates a 'parent' bundle product with 'child' component products
CREATE TABLE product_bundles (
    bundle_id INT REFERENCES products(id),
    component_id INT REFERENCES products(id),
    quantity_needed INT NOT NULL, -- How many components needed for 1 bundle
    PRIMARY KEY (bundle_id, component_id)
);

-- 6. INVENTORY: Tracks stock of a product in a specific warehouse
CREATE TABLE inventory (
    id SERIAL PRIMARY KEY,
    product_id INT REFERENCES products(id),
    warehouse_id INT REFERENCES warehouses(id),
    quantity INT NOT NULL DEFAULT 0,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT quantity_non_negative CHECK (quantity >= 0),
    UNIQUE(product_id, warehouse_id) -- One record per product-warehouse pair
);

-- 7. INVENTORY_HISTORY: Tracks *when* and *why* inventory changed
CREATE TABLE inventory_history (
    id SERIAL PRIMARY KEY,
    inventory_id INT REFERENCES inventory(id),

    change_amount INT NOT NULL, -- Can be positive (restock) or negative (sale)

    reason VARCHAR(50), -- e.g., 'SALE', 'RESTOCK', 'DAMAGE'

    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP

);
```

2. Identifying Gaps (Questions for Product Team)

To build a robust system, I would need clarification on these missing requirements:

1. **Bundle Inventory Logic:** Do we physically stock "Bundles," or is the stock calculation "virtual" (calculated on the fly based on the available components)? My design assumes virtual calculation currently.
2. **Multiple Suppliers:** Can a single product be supplied by multiple vendors at different prices? Currently, I linked one supplier per product for simplicity.
3. **Currency:** Do we need to handle multiple currencies for companies operating internationally?

3. Explanation of Design Decisions

1. **Audit Trail (inventory_history):** The prompt asked to "track when inventory levels change." Instead of just updating the inventory table, I added a history table. This allows the business to generate reports on how fast items are selling or if theft (shrinkage) is occurring.
2. **Data Integrity (CHECK constraints):** I added CHECK (quantity >= 0) on the inventory table. This prevents bugs (like the one in Part 1) from creating impossible negative stock levels in the database.
3. **Performance (Indexes):** While not explicitly written in DDL, I would add indexes on product_id and warehouse_id in the inventory table because those columns will be queried frequently for dashboards.
4. **Money Types:** I used DECIMAL instead of FLOAT for prices to avoid floating-point rounding errors which are critical in financial applications.

Part 3: API Implementation

Problem Summary: The task was to implement an API endpoint (GET /api/companies/{company_id}/alerts/low-stock) using Java Spring Boot. The endpoint must return low-stock alerts based on product thresholds, but only for products with recent sales activity.

1. The Code Implementation (Java Spring Boot)

A. The DTO (Data Transfer Object) First, we define the structure that matches the required JSON output.

Java

```java
// LowStockAlertDTO.java

@Data // Lombok for getters/setters

@Builder

public class LowStockAlertDTO {

    private Long productId;

    private String productName;

    private String sku;

    private Long warehouseId;

    private String warehouseName;

    private int currentStock;

    private int threshold;

    private int daysUntilStockout;

    private SupplierDTO supplier;


    @Data

    @Builder

    public static class SupplierDTO {

        private Long id;

        private String name;

        private String contactEmail;
    }
}
```

B. The Controller & Service This implements the business logic, including the calculation of stock velocity using the inventory history.

Java

```
@RestController
@RequestMapping("/api/companies")
public class InventoryController {

    @Autowired
    private InventoryService inventoryService;

    @GetMapping("/{companyId}/alerts/low-stock")
    public ResponseEntity<Map<String, Object>> getLowStockAlerts(
            @PathVariable Long companyId) {

        List<LowStockAlertDTO> alerts = inventoryService.getLowStockAlerts(companyId);

        Map<String, Object> response = new HashMap<>();
        response.put("alerts", alerts);
        response.put("total_alerts", alerts.size());

        return ResponseEntity.ok(response);
    }
```

```
}

@Service
public class InventoryService {

    @Autowired
    private InventoryRepository inventoryRepo;

    public List<LowStockAlertDTO> getLowStockAlerts(Long companyId) {
        List<LowStockAlertDTO> alerts = new ArrayList<>();

        // 1. Fetch all inventory items for this company where quantity <= threshold
        // (Assuming a custom JPQL query in Repository that joins Product, Warehouse, and
Supplier)
        List<InventoryProjection> lowStockItems =
inventoryRepo.findLowStockItems(companyId);

        for (InventoryProjection item : lowStockItems) {

            // Business Rule: Only alert for products with recent sales
            if (!hasRecentSales(item.getProductId(), 30)) {
                continue;
            }

            // Logic: Calculate Days Until Stockout
            // days = current_stock / avg_daily_sales
            double avgDailySales = getAverageDailySales(item.getProductId(),
item.getWarehouseId());
            int daysUntilStockout = 999; // Default if no sales velocity


            if (avgDailySales > 0) {
                daysUntilStockout = (int) (item.getQuantity() / avgDailySales);
            }


            // Map DB result to DTO
            LowStockAlertDTO alert = LowStockAlertDTO.builder()
                .productId(item.getProductId())
                .productName(item.getProductName())
                .sku(item.getSku())
                .warehouseId(item.getWarehouseId())
                .warehouseName(item.getWarehouseName())
                .currentStock(item.getQuantity())
                .threshold(item.getReorderLevel())
                .daysUntilStockout(daysUntilStockout)
                .supplier(LowStockAlertDTO.SupplierDTO.builder()
                    .id(item.getSupplierId())
                    .name(item.getSupplierName())
                    .contactEmail(item.getSupplierEmail())
                    .build())
                .build();
            alerts.add(alert);
        }
        return alerts;
    }

//  Checks if items were sold recently
    private boolean hasRecentSales(Long productId, int days) {
        // LOGIC: Join tables to find history for this product
        // SQL:
        // SELECT COUNT(*) FROM inventory_history h
        // JOIN inventory i ON h.inventory_id = i.id
        // WHERE i.product_id = :productId
        // AND h.reason = 'SALE'
        // AND h.changed_at > (NOW() - :days)

        // If count > 0, it means we have recent activity.
        return true;
    }

    // Calculates velocity (How many units we sell per day)
    private double getAverageDailySales(Long productId, Long warehouseId) {
        // LOGIC:
        // 1. Identify the specific Inventory Record
        //    SELECT id FROM inventory
        //    WHERE product_id = :productId AND warehouse_id = :warehouseId
        // 2. Sum sales from history for THAT inventory record
        //    SELECT SUM(ABS(change_amount)) FROM inventory_history
        //    WHERE inventory_id = :inventoryId
        //    AND reason = 'SALE'
        //    AND changed_at > (NOW() - 30 days)



        //   return total_sold / 30.0


        return 5.0; // Mock value since we don't have a live DB connection
    }
}
```

2. Assumptions Made (Documenting Gaps)

Since the prompt was incomplete, I made these specific assumptions:

1. **Scope of Alert:** I assumed alerts are generated per warehouse, not globally. If Warehouse A is low but Warehouse B is full, we still alert for Warehouse A to prevent local stockouts.
2. **Recent Activity:** I defined "recent sales activity" as "any sale occurring within the last 30 days."
3. **Stockout Calculation:** I assumed days_until_stockout is a simple linear projection: Current Quantity / Average Daily Sales.

3. Handling Edge Cases

1. **Division by Zero:** If a product is low on stock but hasn't sold recently (avg sales = 0), the math for days_until_stockout would crash. I handled this by defaulting to a safe high number (999) or skipping the division.
2. **Inactive Products:** The code explicitly filters out products with no recent sales to avoid "alert fatigue" for obsolete items that we don't intend to restock.