StockFlow - Inventory Management System for B2B SaaS

Case Study submission for Backend Engineering Intern

Part 1: Code Review & Debugging

Issues in the provided endpoint:

1. **Incorrect transaction handling**
   The Problem: The code calls db.session.commit() twice, once after creating the product and again after adding inventory. This is not atomic. If the second commit fails (e.g. due to DB error), the first commit has already persisted the product, leaving the database in an inconsistent state (product exists without inventory)
   Solution: Perform all related operations in a single transaction and commit once at the end
2. **Business logic violation**
   The Problem: The code sets warehouse_id on Product, implying each product belongs to one warehouse. However, requirements state "Products can exist in multiple warehouses". By tying a product to a single warehouse_id, it fails to allow multi-warehouse inventory tracking. This mis-modeling can lead to inaccurate stock management and inability to track the same product across locations.
   Solution: Instead, a many-to-many Product–Warehouse association (e.g. an Inventory table) should be used
3. **No SKU uniqueness enforcement**
   The Problem: SKUs must be unique across the platform, but the code does not validate or enforce this. Inserting a duplicate SKU would cause a database uniqueness error (if defined) or create duplicate entries, breaking inventory lookup.
   Solution: A UNIQUE constraint on the sku column should be defined. The code should also catch integrity errors to return a user-friendly message if a duplicate SKU is provided.
4. **Price data type**
   The Problem: Price is likely a decimal (money) value, but the code treats it opaquely. If price is stored as a string or float, precision issues may occur.
   Solution: Using a DECIMAL or specialized money type is better to avoid rounding errors. The code should validate price formatting before insertion.
5. **Missing input validation:**
   Problem: The code assumes all fields (name, sku, price,warehouse_id, initial_quantity) are present and valid in data. If any key is missing or has invalid type, this will raise a runtime error. In production, missing or malformed input could crash the endpoint.
   Solution: The code should validate inputs (e.g. check required fields, data types, non-negative initial_quantity) and return a 400 error on invalid data (used marshmallow library).

Impact of each issue:

1. **Non-atomic commits**: As discussed, if the second commit fails (e.g. due to a constraint violation), the first part (creating the product) remains committed. This "half success" scenario yields inconsistent state: inventory not initialized for a product, potentially causing orphaned products or negative stock.

2. **Single-warehouse modeling**: This violates requirements and will cause bugs when inventory is moved or the product needs to be stocked in a new warehouse. It prevents recording different quantities in each warehouse. Inventory reports and replenishment logic will fail to account for multi-warehouse.
3. **Duplicate SKU risk**: Without handling, a duplicate SKU insertion will cause an exception (or duplicate logical entry). In either case, this leads to data corruption or a 500 error. Uniqueness should be enforced at the DB level and handled in code to avoid silent corrupton.
4. **Incorrect price handling**: Using a float for monetary values can lead to precision loss. In financial domains, this is unacceptable; order totals and accounting reports would be off. A Decimal type with validation avoids such rounding issues.
5. **Missing validation**: Unchecked inputs can lead to KeyError or type errors, crashing the service. For example, if initial_quantity is omitted, the code throws an exception and no response is returned, causing an HTTP 500. Input validation prevents this by returning a controlled error response.

Code Quality:

1. Input Validation: Implemented Marshmallow schema (ProductSchema) for input validation
2. Error Handling: Added centralized error handlers for ValidationError and IntegrityError
3. Constants: Defined constants for HTTP status codes and error messages
4. Type Hints: Added type hints to function signatures
5. Logging: Added logging configuration and log statements for key events
6. Standardized Responses: Created a create_response helper function for consistent API responses
7. Documentation: Added detailed docstring to the endpoint

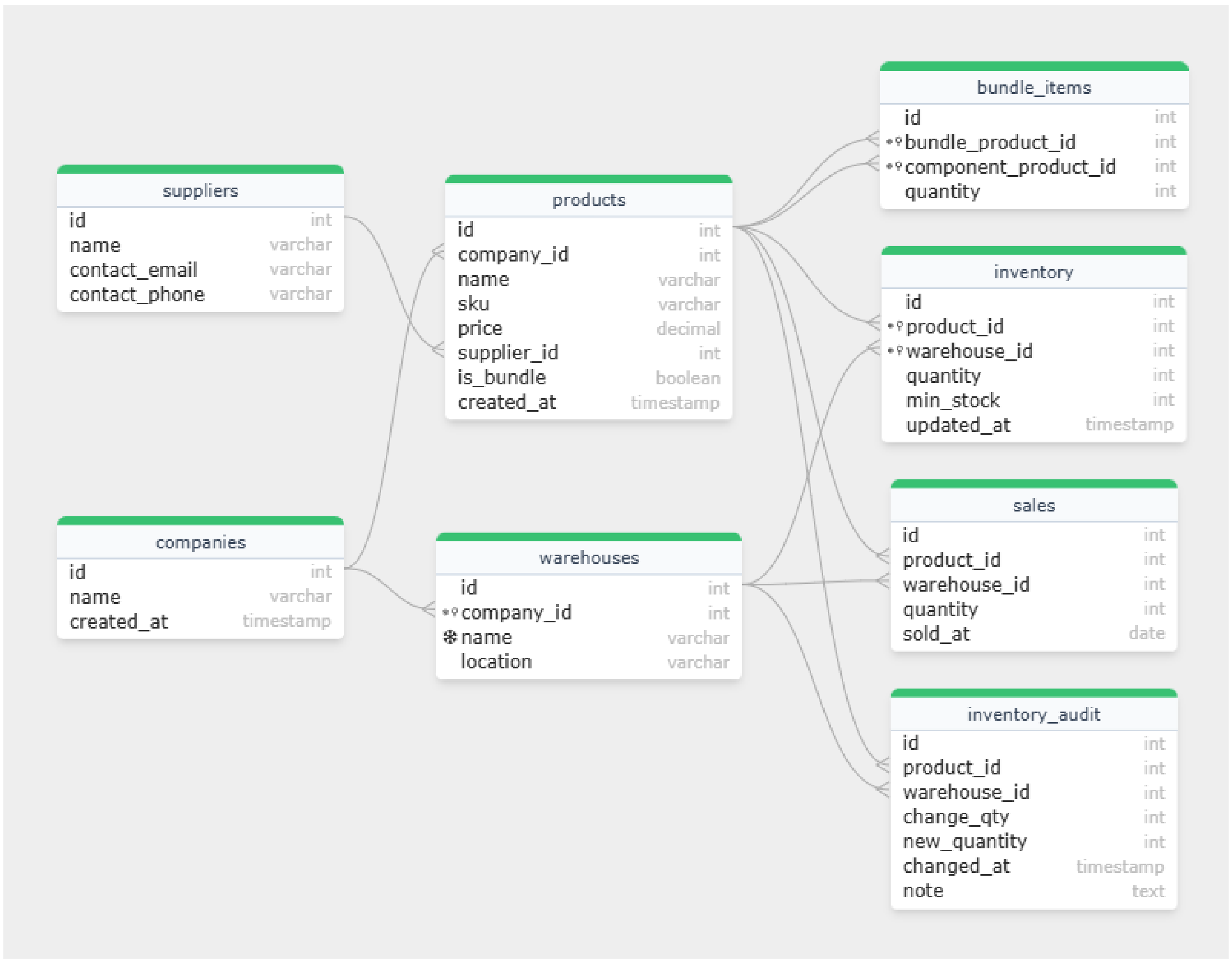
Corrected implementation:

Used **Postman** to test API.

All operations are performed in one transaction, inputs are validated, and errors are handled. The Product model no longer includes warehouse_id (inventory is separate). I have assumed SQLAlchemy models with relationships.

View code on Github

Part 2: Code Review & Debugging



Design justifications:

1. **Normalization**: Each table captures one entity and its attributes to avoid. E.g. product attributes are kept in products; warehouse details in warehouses; supplier info in suppliers. This ensures no duplicate storage of e.g. product names or supplier details.
2. **Inventory as junction table**: The inventory table resolves the many-to-many between products and warehouses, allowing any product to be in multiple warehouses with its own quantity. Each record also stores data like min_stock (reorder point).
3. **Foreign keys & constraints**: I used foreign key constraints (with ON DELETE CASCADE) to maintain referential integrity. E.g. deleting a company cascades to its warehouses. The UNIQUE constraint on (product_id, warehouse_id) prevents duplicate inventory entries. The sku field has a UNIQUE constraint to enforce SKU uniqueness
4. **Indexes:** Primary keys are indexed by default. We should index foreign key columns for performance, as joins on these columns are frequent. For example, inventory(product_id) and sales(sold_at) might be indexed to speed common queries (e.g. finding recent sales). Indexing sales(sold_at) also helps date-range queries for recent sales.
5. **Bundle modeling:** We represent bundles in products (with is_bundle flag) and link components in bundle_items. This normalized approach avoids duplicating component attributes in every bundle and allows flexible bundle definitions.

6. **Audit log**: inventory_audit records every stock change with timestamp and optional note. This full history is critical for tracking when inventory levels change (a given requirement). It could grow large, but it’s append-only and can be partitioned or archived if needed.
7. **Sales:** The sales table logs item sales with date and quantity. This is used by the low-stock alert logic (recent sales). We keep it simple (one row per product sold per day); in a real system, an order/line-item table might be used.

Missing requirements / questions:

Some areas need clarification or additional info, such as:

- Thresholds: How are low-stock thresholds defined? By product category, as raw number, or dynamic (e.g. days-of-stock)? Must thresholds be configurable per product or type?
- Recent sales window: What qualifies as “recent sales activity” (last 30 days? 90 days?). How to handle products with sporadic sales?
- Bundle sales handling: If a bundle is sold, should components’ inventory be decremented? How are bundles accounted for in alerts?
- Supplier mapping: Can a product have multiple suppliers? If so, how do we pick the correct one for reordering in alerts? Should we store a preferred supplier?
- Warehouse policies: Are there warehouse-specific reorder levels? Can different warehouses for same product have different min_stock? (Our design allows it via inventory.min_stock.)
- Sales granularity: Should sales include timestamps, channel info, or just date/quantity? Are returns handled?
- User roles and security: (Not specified) How to handle who can create inventory or view alerts?

Part 3: API Implementation

(Used Postman and SQLalchemy)

Assumptions

For this endpoint, we assume:

1. "Recent sales" means sales in the last 30 days.
2. Each product has a type or flag that determines its low-stock threshold. For illustration, we assume standard products have threshold = 20 units, bundles = 10 units. (In practice, thresholds might come from a config or table.)
3. Thresholds and sales averages are computed per warehouse. That is, inventory and sales are evaluated per warehouse_id.
4. Each product has a single primary supplier (stored in product.supplier_id). We include that supplier's info in the alert for reorder.
5. If a product has no sales in the period, we do not generate an alert (no recent activity).
6. If average daily sales is zero (no sales), we skip that product from alerts.
7. days_until_stockout = current_stock ÷ average_daily_sales (rounded down).

Logic explanation:

- We join Inventory->Product->Warehouse (filtered by company) and optionally Supplier to get all needed fields.
- We loop through each product-inventory record. We aggregate sales for that product over the last 30 days. If sales_sum == 0, we skip, enforcing "recent sales activity" rule.
- We compute daily_avg = sales_sum / 30. The low-stock **threshold** is chosen by product type (here, bundle vs. standard). If current stock is below threshold, we create an alert.
- days_until_stockout is current_stock / daily_avg (rounded down). If daily_avg is zero (edge case), we set this to None or omit it (meaning stockout is indeterminate).
- Each alert includes supplier info (id, name, contact_email) for reorder. If no supplier is set, those fields are null.

Edge case handling:

- No sales: Products with zero sales in the period produce no alert (they're ignored). We assume no reorder needed if none have sold.
- Zero average sales: If daily_avg is zero (e.g. all sales occurred on one day), we avoid division by zero. In our code, days_until_stockout becomes None. Alternatively, could treat this as 'infinite' days.
- Missing supplier: If a product has no linked supplier, we still include it in alerts but supplier fields will be null. The client should handle missing supplier gracefully.
- Threshold defaults: If a product's type isn't recognized, we default to 'standard' threshold. In real code, thresholds could be stored in the DB or config.

- Multiple warehouses: We check inventory per warehouse. If a company has multiple warehouses, the alert covers each warehouse separately. (E.g. if a product is low in one warehouse but not in another, only the low one appears in results.)