NAME - Shruti Prashant Lad shrutilad35@gmail.com

Part 1: Code Review & Debugging

Original Problem

The existing implementation for creating products works at compile time but fails in production due to **design, transactional, and business-logic issues**. The system must support:

- Products in multiple warehouses
- Globally unique SKUs
- Decimal pricing
- Optional inventory creation

1. Issues Identified

Issue 1: Product incorrectly coupled with Warehouse

- The product entity contains `warehouseId`.

Problem:
Products can exist in **multiple warehouses**, so the relationship must be handled via an **Inventory** table.

Issue 2: SKU uniqueness not enforced

- No validation or DB constraint exists.

Problem:
Duplicate SKUs break order processing, reporting, and integrations.

Issue 3: Missing transaction management

- Product and Inventory are saved separately.

Problem:
If inventory creation fails, the product remains saved → **inconsistent data**.

Issue 4: No request validation

- Assumes all fields are present.

Problem:
Leads to runtime exceptions and unstable APIs.

Issue 5: Price precision not handled

- Monetary values stored as `float/double`.

Problem:
Causes rounding issues in billing and accounting.

Issue 6: Inventory creation is mandatory

- Inventory is always created.

Problem:
Products may be created before being stocked.

Issue 7: No tenant (company) validation

- No company scoping present.

Problem:
In multi-tenant SaaS, this risks **data leakage**.

2. Impact in Production

Issue and Impact

1. Product-Warehouse coupling - Cannot scale to multiple warehouses
2. Duplicate SKUs - Order & inventory conflicts
3. Partial commits - Orphaned product records
4. No validation - Orphaned product records

5. Floating-point price - Financial inaccuracies

3. Corrected Implementation (Java – Spring Boot)

Key Improvements

✓ Uses `@Transactional`
✓ Decouples Product from Warehouse
✓ Enforces SKU uniqueness
✓ Uses `BigDecimal` for pricing
✓ Handles optional inventory creation

Service Layer (Corrected Code)

```
@Service
public class ProductService {

    private final ProductRepository productRepository;
    private final InventoryRepository inventoryRepository;

    public ProductService(ProductRepository productRepository,
                          InventoryRepository inventoryRepository) {
        this.productRepository = productRepository;
        this.inventoryRepository = inventoryRepository;
    }

    @Transactional
    public Long createProduct(CreateProductRequest request) {

        // Validation
        if (request.getName() == null || request.getSku() == null ||
request.getPrice() == null) {
            throw new IllegalArgumentException("Name, SKU, and Price
are required");
        }

        // SKU uniqueness check
        if (productRepository.existsBySku(request.getSku())) {
```

```
            throw new IllegalStateException("SKU already exists");
        }

        // Create Product (warehouse-independent)
        Product product = new Product();
        product.setName(request.getName());
        product.setSku(request.getSku());
        product.setPrice(request.getPrice()); // BigDecimal

        productRepository.save(product);

        // Create inventory only if warehouse is provided
        if (request.getWarehouseId() != null) {
            Inventory inventory = new Inventory();
            inventory.setProduct(product);
            inventory.setWarehouseId(request.getWarehouseId());
            inventory.setQuantity(
                request.getInitialQuantity() != null ?
request.getInitialQuantity() : 0
            );
            inventoryRepository.save(inventory);
        }

        return product.getId();
    }
}
```

4. Explanation of Fixes

Transaction Safety

- `@Transactional` ensures **atomicity** (all-or-nothing).

Multi-Warehouse Support

- Product is warehouse-agnostic.
- Inventory maps product ↔ warehouse.

SKU Uniqueness

- Enforced at service + database level.

Financial Accuracy

- `BigDecimal` used for all price values.

Optional Inventory

- Supports real-world workflows where stock arrives later.

Conclusion

This Java-based solution:

- Follows **Spring Boot best practices**
- Prevents data inconsistencies
- Supports scalability and multi-warehouse operations
- Aligns with real-world B2B SaaS requirements